Depurador algorítmico para estructuras de control Java

Javier Alejos Castroviejo
Diego Arranz García
Saskya Isabel Mosquera Logroño

Proyecto de Sistemas Informáticos,
Facultad de Informática,
Departamento de Sistemas Informáticos,
Universidad Complutense de Madrid

Madrid, 6 de Junio de 2014

Director : Rafael Caballero Roldán
Codirector : Adrián Riesco Rodríguez

Autorización de difusión y utilización

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluye imágenes de los autores, la documentación y/o el prototipo desarrollado

FDO: JAVIER ALEJOS CASTROVIEJO, EN MADRID 6 DE JUNIO DE 2014

FDO: DIEGO ARRANZ GARCÍA, EN MADRID 6 DE JUNIO DE 2014

FDO: SASKYA ISABEL MOSQUERA LOGROÑO, EN MADRID 6 DE JUNIO DE 2014

Agradecimientos

En primer lugar queremos agradecer en especial a nuestras familias y amigos por habernos apoyado, por toda la paciencia y comprensión, no solo durante el proyecto, sino durante toda nuestra carrera.

También queremos agradecer la enorme labor de nuestro director de proyecto Rafael Caballero Roldán y al codirector Adrián Riesco Rodríguez por sus aportaciones.

Sin todos ellos no hubiese sido posible haber llegado hasta aquí.

Índice

Índice de figuras

Figuras 2 : Análisis y diseño de la aplicación

Figuras 3 : Funcionamiento de la aplicación

Resumen

Este proyecto presenta un generador de trazas de programas Java destinado a localizar errores en el código. Inicialmente se describe *Java Platform Debugger Architecture* (JPDA) y *Java Debugger Interface* (JDI) que son las bases de la aplicación. Más adelante, se profundiza en la estructura de la aplicación, mostrando los detalles más técnicos y también las dificultades encontradas durante el desarrollo del proyecto, y las medidas aplicadas para resolverlas. Además, se muestra el funcionamiento de la aplicación y sus limitaciones. A continuación, se concluye mostrando unos casos de prueba con los que se ha comprobado el correcto funcionamiento. Finalmente un apartado de conclusiones que se han obtenido tras la realización del proyecto y un posible trabajo futuro para la ampliación del mismo.

Abstract

In the project explained below, we have designed a trace generator for Java programs in order to facilitate the debugging phase. First, we describe the Java Platform Debugger Architecture (JPDA) and the Java Debugger Interface (JDI), which constitute the basis of the application. Second we go into the application structure, showing the technical details and the difficulties found during the project and the measures taken to resolve them. Furthermore, we show the application behaviour and its limitations. Next we present some test cases that confirm the correctness of our approach. Finally, we discuss the conclusions that have been obtained after the completion of the project and possible future extensions of the application.

Palabras clave

Depurador, Java, *Java Debbugger Interface* (JDI), *Java Platform Debugger Architecture* (JPDA), Traza, Reflexión.

Keywords

Debug, Java Debbugger Interface (JDI),Java, Java Platform Debugger Architecture (JPDA), Profile, Reflection, Trace.

1. Introducción

El propósito de este proyecto es el desarrollo de una herramienta que permita visualizar cómputos de programas Java. Se desea mostrar en forma de árbol la jerarquía de llamadas entre métodos en una ejecución concreta con el propósito de analizar, entender y depurar el programa. Para ello se presentan a continuación los términos más relevantes para comprender el alcance del proyecto.

Java

Java es un lenguaje de programación orientado a objetos que permite concurrencia. Fue creado inicialmente en la década de los 90 con el eslogan "escribe una vez, ejecuta donde quieras". La idea detrás de este pensamiento es llevar el código a un lenguaje intermedio, los *bytecodes*, interpretados por la Java Virtual Machine (JVM). Así cualquier sistema operativo o dispositivo que tenga instalada la JVM puede ejecutar el mismo código compilado.

La sintaxis de Java se asemeja a la de C y C++, pero el modelo de ejecución es muy distinto, en particular a la gestión de memoria, ya que Java dispone de la llamada "recogida automática de basura", que permite al programador obviar la compleja (y propensa a errores) tarea de liberación de memoria dinámica de otros lenguajes. [1]

Se ha escogido Java como lenguaje de programación porque Java es un tipo de lenguaje que soporta la reflexión. La reflexión es la capacidad de un lenguaje de programación que consiste en poder observar e incluso manipular instancias de objetos de un programa en tiempo de ejecución.

Además se trata posiblemente del lenguaje de programación más difundido, debido a sus muchas ventajas, entre las cuales tenemos que se trata de un lenguaje multiplataforma, lo que quiere decir que se ejecuta en la mayoría de los sistemas operativos, inclusive en sistemas operativos móviles. Otra de las ventajas es que Java es un software de distribución libre, no es necesario pagar una licencia para poder comenzar a desarrollar en este lenguaje. Asimismo es un lenguaje muy completo y versátil, pues posee multitud de bibliotecas y utilidades muy completas que facilitan la programación [2].

Depuración

Se llama depuración de programas al proceso de búsqueda y corrección de errores en programas y sistemas informáticos. En inglés se conoce como *debugging*, según la leyenda que relata que en los primeros tiempos consistía realmente en la eliminación de bichos (*bugs*), que producían cortocircuitos en las válvulas de vacío de los primeros ordenadores.

De hecho esta historia es controvertida. Aunque los términos *bug* y *debugging* son atribuidos popularmente a la almirante Grace Murray Hopper por los años 1940 y asociados a la Mark II de la Universidad de Harvard, el término *bug* cómo significado de error técnico data cerca de 1878, y el término *debugging* o depuración fue usado en aeronáutica antes de entrar al mundo de las computadoras.

La mayoría de los depuradores existentes en la actualidad se basan en la ejecución paso a paso del programa. Estos depuradores se basan en pausar el programa para examinar el estado actual en cierto evento o instrucción especificada por medio de un *breakpoint*, o ejecutando las instrucciones una a una, y en el seguimiento de valores de algunas variables. Se puede decir que mientras que en el desarrollo de software ha habido diversos cambios de paradigma (por ejemplo, la introducción de la programación orientada a objetos, nuevas técnicas de análisis y metodologías de desarrollo), en el caso de la depuración se siguen utilizando las mismas técnicas que hace 40 años. Analizando el modelo de depuradores “paso a paso” se puede decir que es una tarea que consume mucho tiempo, es compleja y sobre todo tediosa y no ha variado en los últimos años.

En la ingeniería del software se proponen métodos para disminuir la posibilidad de error y también para probar (casos de prueba), pero faltan propuestas de “alto nivel” para depurar, ya que a pesar de que el programa compile, algunas veces da como resultado algo que no se esperaba. Entonces empieza una de las tareas más difíciles para los programadores que es la de identificar y subsanar los errores.

Objetivos

Como consecuencia de esto surge la necesidad de crear un nuevo sistema de depuración, que se basa en generar una traza completa de la secuencia del programa que estamos depurando, y donde el objetivo fundamental es generar un árbol con las llamadas que se han realizado durante el flujo de todo el mismo, que permite optimizar en tiempo y recursos, sintetizando de esta forma su presentación, consiguiendo una forma fácil de interpretar.

Herramienta Inspector que permita visualizar una ejecución a un alto nivel representada por una estructura de árbol. En dicha estructura, los nodos corresponden a llamadas a métodos, la raíz representa la llamada a método y la relación padre-hijo entre varios nodos indica la invocación de un método (el hijo) dentro de otro (el padre). (Ver capítulo 2, para una descripción más precisa).

La aplicación está orientada a Java. Al abstraer detalles del código no hace falta tener el código fuente original; se desea que pueda utilizarse sobre un JAR o sobre ficheros class.

Esto corresponde al objetivo inicial, con el que se empezó a trabajar en el proyecto, durante el transcurso del mismo, se encontró un inconveniente, que era el gran número de nodos que generaba, provocando un árbol demasiado extenso, con información que muchas veces no era de utilidad para realizar una depuración fácil y rápida. Esto motiva el segundo objetivo:

Herramienta Profiler que muestre la "carga" en el sentido de número de llamadas por clases y métodos. El objetivo en relación con el *inspector* es que el *profiler* permita eliminar tanto métodos como clases y paquetes en los que se confíe, aligerando la tarea del inspector y el número de nodos.

Aunque con esta mejora, se intentó acotar más el espacio de depuración, dio un motivo para plantearse esta nueva herramienta:

Herramienta Settings, aquí es donde se pueden manipular las distintas configuraciones, donde no solo se va a poder excluir métodos, clases o paquetes desde la herramienta *profiler*, sino que también se va a poder decidir si se excluye el this, bibliotecas o estructuras de datos, lo que va a permitir también aligerar la tarea del inspector.

En el siguiente capítulo se presenta el diseño y análisis de la aplicación. Se comentan las dificultades encontradas y la aplicación que se ha dado a los conocimientos adquiridos durante los estudios que concluyen con este trabajo. El capítulo 3 describe la aplicación: su funcionamiento, aplicaciones y limitaciones. Se muestra un ejemplo de aplicación. Finalmente, el capítulo 4 comenta las conclusiones y posible trabajo futuro. Además esta memoria incluye un apéndice con diversos casos de uso que muestran las posibilidades de la propuesta.

Bibliografía

[1] Java, https://www.java.com/es/download/whatis_java.jsp
[2] Thinking in Java, Bruce Eckel, 4ª edición, Prentice Hall, 2006.

2. Análisis y diseño de la aplicación

En este capítulo se mostrará una visión general y técnica sobre el diseño de la aplicación, como pueden ser la estructura de la aplicación, los diagramas de flujo, la evolución de la aplicación, etc. Las secciones de este capítulo son:

1. **Java Platform Debugger Architecture y Java Debugger Interface:** Información previa que es necesaria para comprender la base sobre la que opera la aplicación.

2. **Visión general de la aplicación:** Se muestra una idea general de la aplicación y de las dimensiones del proyecto.

3. **Bibliotecas utilizadas:** Se describe todas aquellas bibliotecas utilizas en el proyecto.

4. **Estructura de la aplicación:** Se describe los aspectos técnicos en profundidad para poder entender el proyecto.

5. **Dificultades encontradas:** Se muestran una serie de problemas que surgieron durante el desarrollo del proyecto y las medidas que se tomaron a cabo.

6. **Etapas del proyecto:** Se describe cronológicamente la evolución del proyecto.

7. **Conocimientos empleados:** Se mencionan los conocimientos que se han adquirido a lo largo de la carrera y se han ido empleado en el proyecto.

2.1. Java Platform Debugger Architecture y Java Debugger Interface

Java Platform Debugger Architecture (JPDA) es la arquitectura sobre la cual trabaja al aplicación, por ello a continuación se explicará con cierto detalle a continuación para un mejor entendimiento de la aplicación.

JPDA consta de tres interfaces diseñadas para uso de los depuradores en entornos de desarrollo para sistemas de escritorio. *Java Virtual Machine Tools Interface* (JVM TI) define los servicios que la máquina virtual debe proporcionar para la depuración. *Java Debug Wire Protocol* (JDWP) define el formato de la información y las solicitudes que se transfieren entre el proceso que se está depurando y el depurador, que implementa la *Java Debugger Interface* (JDI). *Java Debugger Interface* define la información y las solicitudes a nivel de código de usuario [1]:

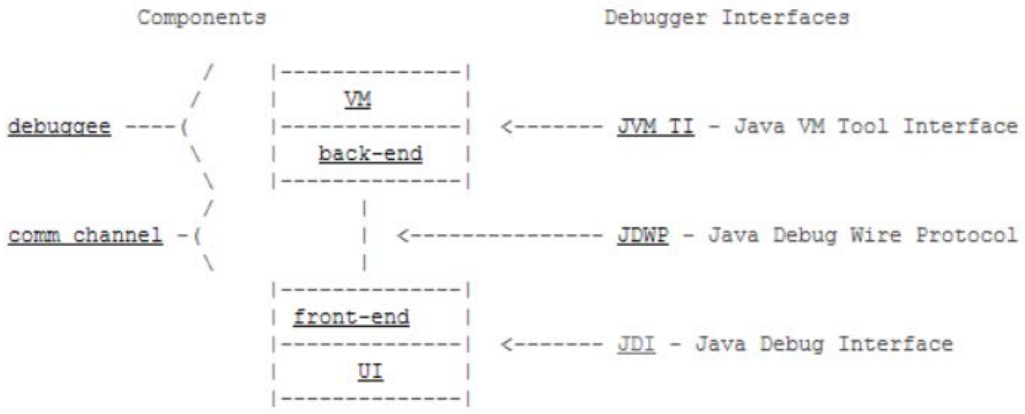


Figura 2.1: *Java Platform Debugger Architecture Components*

Las componentes que constituyen la arquitectura JPDA observadas en la figura 2.1 son numerosas, a continuación se describe brevemente el cometido de cada una de ellas [2]:

- ***Debuggee*:** Es el proceso que se está depurando, que consta de la aplicación que se está depurando, la máquina virtual que ejecuta la aplicación y el *back-end* del depurador.

- ***Java Virtual Machine* (JVM o VM):** Es la máquina virtual que ejecuta la aplicación que se está depurando. La arquitectura depurador está diseñada para su uso en una amplia gama de implementaciones. La VM implementa la *Java Virtual Machine Tool Interface* (JVM TI).

- ***Back-end*:** El *back-end* del depurador es responsable de comunicar las solicitudes del *front-end* depurador al depurador VM y de comunicar la respuesta a estas solicitudes (incluyendo eventos deseados) al *front-end*. El back-end se comunica con el front-end a través de un canal de comunicaciones utilizando *Java Debug Wire Protocol* (JDWP), mientras que el back-end se comunica con el código depurado utilizando la *Java Virtual Machine Tool Interface* (JVM TI).

- **Canal de comunicaciones:** El canal de comunicación es el vínculo entre el *back-end* y *front-end* del depurador. Se puede considerar que consta de dos mecanismos:

    - **Un conector:** Un conector es un objeto JDI que es el medio por el que se establece una conexión entre el *back-end* y *front-end*. JPDA define tres tipos de conectores:

        - **Conectores de escucha (*Listening connectors*):** El front-end a la escucha de una conexión entrante desde el *back-end.*

        - **Conectores de ajustamiento (*Attachment connectors*):** El *front-end* se une a una ya en ejecución en *back-end*.

        - **Conectores de lanzamiento (*Launching connectors*):** El *front-end* pone en marcha el proceso de Java que ejecutará el código depurado y el *back-end*.

    - **Un transporte:** Un transporte es el mecanismo subyacente que se utiliza para mover bits entre el *front-end* y el *back-end*. El mecanismo de transporte utilizado no está especificado, no obstante los distintos tipos transporte incluyen mecanismos como *sockets*, líneas de serie, y memoria compartida. Sin embargo, el formato y la semántica del flujo de bits en serie que fluye a través del canal se especifica mediante *Java Debug Wire Protocol* (JDWP).

- ***Front-end*:** El *front-end* depurador implementa a alto nivel la *Java Debugger Interface* (JDI). El *front-end* utiliza la información del bajo nivel de *Java Debug Wire Protocol* (JDWP).

Para entender de manera más práctica dicha estructura, Oracle ofrece en su página web una serie de ejemplos de aplicaciones básicas utilizando JPDA. En el caso de JavaTracer necesitaba comunicarse únicamente a través *Java Debugger Interface*, ya que no hay que declarar nuevos servicios (JVM TI) ni tampoco modificar la capa de transporte (JDWP). Entre todas estas aplicaciones, el proyecto se basó en una de las tres aplicaciones que ofrece Oracle sobre JDI, Tracer[5]. Esta aplicación mostraba un esqueleto básico sobre cómo utilizar el *launching connector* y el funcionamiento de la generación de eventos que más adelante se verá en detalle.

Primero se debe encontrar y configurar el conector que se utilizará para hacer de conexión entre el *front-end* y el *back-end*. En el caso de JavaTracer se ha usado el conector *CommandLineLaunch* que permite la ejecución de aplicaciones java introduciendo un comando como si se estuviese haciendo a través de consola, por lo que este conector pertenecería al grupo de los *launching connector*. El siguiente trozo de código proporciona una visión más práctica de cómo usar dicho conector.

```
/* Primero deberemos encontrar el conector necesitado */

LaunchingConnector findLaunchingConnector() {
        List<Connector> connectors =
                            Bootstrap.virtualMachineManager().allConnectors();
        for (Connector connector : connectors) {
               if (connector.name().equals("com.sun.jdi.CommandLineLaunch")) {
                      return (LaunchingConnector) connector;
               }
        }
}

/* Se configura el conector con las opciones que se deseen, las opciones se
encontrarán recogidas en un Map<String,Argument> para más tarde lanzar el conector
con dichas opciones. Para configurar correctamente el conector habrá que elegir una
opción main (Clase elegida con un método main o un jar ejecutable). Primero se
recogen los argumentos por defecto del conector */

       Map<String, Connector.Argument> arguments = connector.defaultArguments();

/* Se obtienen los objetos de los argumentos main y option */

       Connector.Argument mainArg = (Connector.Argument) arguments.get("main");
       Connector.Argument optionArg = (Connector.Argument) arguments.get("options");
```

/* En el caso de que sea un fichero class, lo que hay que hacer es introducir como argumento main el nombre de la clase donde se comenzará la aplicación. Java por defecto busca en el *classpath* las clases que va a utilizar para la ejecución. Por lo que si se quiere ejecutar un fichero class desde cualquier ruta se tendrá que actualizar dicho classpath. Esto es recomendable realizarlo con la opción -classpath(-cp abreviada) para así no cambiar el classpath permanentemente. El comando será:

java -cp *ruta_class nombre_class*

*/

```
String main = nombre_class;
String options = "-cp " + ruta_class;
```

/* A través de la línea de comandos, un jar se lanza de la siguiente manera:

java -jar *nombre_jar*

Por lo que hay que configurar los argumentos para que coincidan con el comando, es decir, en el argumento main se introduce la ruta de dicho jar y en el argumento option -jar. */

```
String main = ruta_jar;
String options = "-jar";
```

/* Ahora solo faltaría por añadir los String que se han formado para el lanzamiento del comando ya sea un fichero jar o class. */

```
mainArg.setValue(main);
optionArg.setValue(options);
```

/* En el caso de que se desee incorporar algún argumento al inicio de la aplicación (args), habrá que añadir el valor concatenandolo con el argumento main. Por ejemplo el argumento para la ejecución de una clase con dos argumentos sería:

java -cp *ruta_class nombre_class argumento1 argumento2*

*/

```
String main = nombre_class + argumento1 + " " + argumento2;
```

/* En el caso que se quiera añadir bibliotecas externas al programa se deben de añadir al classpath. Para añadir correctamente los directorios estos deben estar separados por el separador que depende del sistema operativo. Un ejemplo sería:

java -cp *ruta_class*;*ruta_biblioteca1*; *nombre_class*

*/

```
String option = "-cp " + ruta_class + File.separator + ruta_bibloteca1 + File.separator;
```

Ya configurado el conector, el siguiente paso es lanzarlo con el fin de obtener la *VirtualMachine* asociada al *debuggee*.

```
/*
      Map<String,Argument> arguments = configureConnector(conector);
      VirtualMachine vm  = connector.launch(arguments);
*/
```

Una vez obtenida la *VirtualMachine* habrá que habilitar los eventos que se quiera recibir.

```
/* Hay que obtener el EventRequestManager asociado a la VirtualMachine para configurar los eventos que se vayan a utilizar. Existe una peculiaridad a la hora de configurar la política de suspensión de los distintos managers.Hay tres políticas:
```

- **SUSPEND_ALL:** Suspende la máquina virtual entera suspendiendo todos los threads asociados a ella.
- **SUSPEND_THREAD:** Suspende el thread que ha generado el evento.
- **SUSPEND_NONE:** No suspende nada cuando ocurre el evento.

```
Es importante mencionar que no se puede acceder a ningún tipo de variable u objeto si no está suspendido o bien el thread o bien la máquina virtual. Hay numerosos tipos de eventos, pero en el siguiente código se muestra brevemente cómo configurar el registro de eventos de excepciones. */

      EventRequestManager mgr = vm.eventRequestManager();
      ExceptionRequest excReq = mgr.createExceptionRequest(null,true, true);
      excReq.setSuspendPolicy(EventRequest.SUSPEND_ALL);
      excReq.enable();
```

Una vez se han configurado todos los tipos de eventos, faltaría la forma de recoger dichos eventos. Para ello hay que utilizar una *EventQueue* asociada a una *VirtualMachine* la cuál va almacenando los eventos según van ocurriendo respetando el orden.

```
/* Este trozo de código muestra un ejemplo de cómo recoger los eventos hasta que finalice el programa (variable connected que almacena si estamos aún conectados a la VirtualMachine) */

	EventQueue queue = vm.eventQueue();
	while (connected) {
		try {

		EventSet eventSet = queue.remove();
		EventIterator it = eventSet.eventIterator();
		while (it.hasNext())
			handleEvent(it.nextEvent());
		eventSet.resume();

		} catch (InterruptedException exc) {
			exc.printStackTrace();
		} catch (VMDisconnectedException discExc) {
		discExc.printStackTrace();
			break;
		}
	}
```

2.2. Visión general de la aplicación

La aplicación se ha desarrollado con el objetivo de tener la mayor modularidad posible, para facilitar la extensión y el mantenimiento de la misma. Por ello, se diseñó entorno a tres módulos independientes entre sí (**Profiler**, **Tracer** e **Inspector**). No obstante, estos tres módulos están comunicados a través de un cuarto módulo (**General**) el cuál permite la interacción entre los distintos módulos. De esta manera se logró dotar a la aplicación de una alta usabilidad, permitiendo al usuario navegar entre las distintas aplicaciones. En la siguiente figura se muestra un diagrama UML de los módulos mencionados.

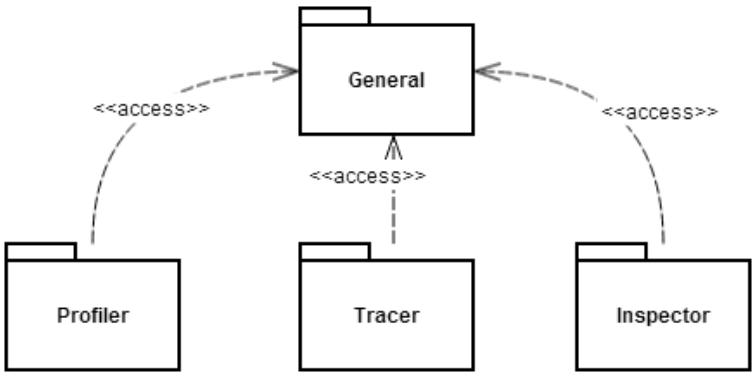


Figura 2.2: Diagrama UML de los distintos módulos del proyecto

Casos de uso

En Figura 2.3 se detalla gráficamente los distintos casos de uso de la aplicación que son descritos a continuación:

- **Generar traza:** Se genera una traza de un programa Java seleccionado por el usuario.
- **Generar profile:** Se genera un profile de un programa Java seleccionado por el usuario.
- **Inspeccionar traza:** El usuario puede inspeccionar una traza seleccionada.
- **Inspeccionar profiler:** El usuario puede abrir un profile seleccionado.
- **Configurar aplicación:** El usuario puede configurar las distintas opciones de la aplicación.

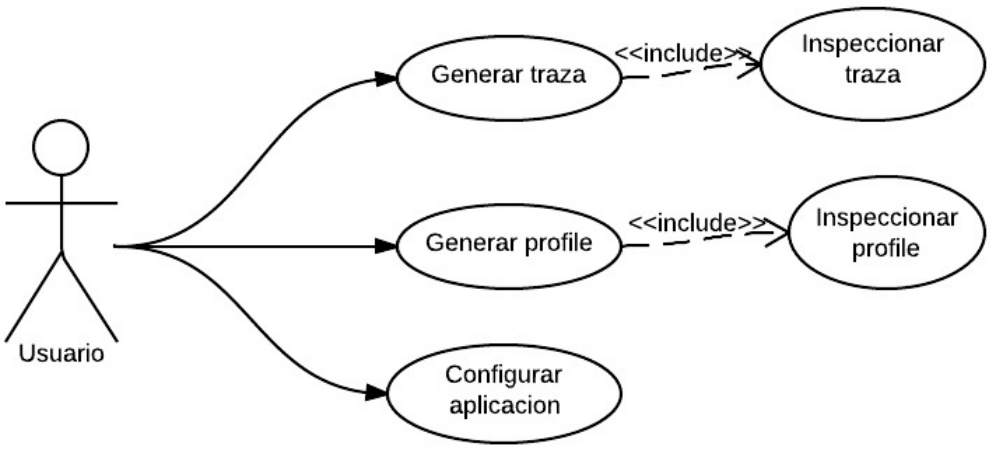


Figura 2.3: Descripción de casos de uso de la aplicación

Métricas

Antes de hablar sobre aspectos técnicos de la aplicación, se mostrarán una serie de métricas para dar a conocer las características del proyecto y así tener una visión más amplia sobre sobre sus dimensiones. Estas métricas han sido tomadas con el plugin Metrics para *Eclipse* [3,4]:

- **Número de paquetes:** Es el número de paquetes definido en el proyecto.
- **Número de clases:** Es el número de clases definidas en el proyecto.
- **Número de clases hijas:** Es el número total de clases definidas que heredan clases implementadas en el proyecto. También las clases que implementan interfaces definidas cuenta como clase hija.
- **Número de interfaces:** Es el número de interfaces definidas en el proyecto.
- **Número de métodos:** Es el número de métodos definidos en el proyecto.
- **Número de métodos estáticos:** Es el número de métodos estáticos definidos en el proyecto.
- **Número de atributos:** Es el número total de atributos definidos en las clases del proyecto.

- **Número de atributos estáticos:** Es el número de atributos estáticos definidos en el proyecto.

- **Número de líneas de código:** Es el número de líneas de código sin incluir líneas en blanco y líneas comentadas.

- **Líneas de código en métodos:** Son el número de líneas de código dentro de un método (no cuenta líneas en blanco ni comentarios dentro del cuerpo de los métodos).

- **Profundidad del árbol de herencia:** Es la distancia (nivel de profundidad) media respecto de la clase Object en la jerarquía de herencia.

- **Nivel de acoplamiento aferente:** Es el número medio de clases fuera de un paquete que dependen de las clases dentro del paquete.

- **Nivel de acoplamiento eferente:** Es el número medio de clases dentro de un paquete que dependen de las clases fuera del paquete.

- **Abstracción:** Es el número de clases abstractas (e interfaces) dividido por el número total de tipos en un paquete.

- **Instanciabilidad:** La instanciabilidad se calcula como Ce / (Ca + Ce), donde Ce es el nivel de acoplamiento eferente y Ca es el nivel de acoplamiento aferente.

- **Distancia normalizada:** La distancia normalizada se calcula como | A + I - 1 |, donde A es la abstracción e I es la instanciabilidad. Este número debe ser cercano a 0 indicando de esta manera el buen diseño de paquetes en la aplicación.

- **Escasez de cohesión en métodos:** Es una métrica para determinar el nivel de cohesión de una clase. Si m(A) es el número de métodos que acceden a un atributo A, la escasez de cohesión se calcula mediante el promedio de m(A) para todos los atributos, se resta el número de métodos m y se divide el resultado por (1-m). Un valor bajo indica una clase cohesionada y un valor cercano a 1 indica una falta de cohesión y sugiere la clase podría mejor ser dividida en subclases.

- **Índice de especialización:** Es una métrica a nivel de clase que se define como NORM * DIT / NOM, donde NORM es el número de métodos sobreescritos, DIT es la profundidad del árbol de herencia y NOM es el número de métodos. Esta métrica muestra en qué medida las subclases redefinen el comportamiento de sus superclases [6]:

Métrica	Valor
Número de paquetes	33
Número de clases	94
Número de clases hijas	16
Número de interfaces	21
Número de métodos	1045
Número de métodos estáticos	14
Número de atributos	406
Número de atributos estáticos	193
Número de líneas de código	9333
Líneas de código en métodos	5495
Profundidad del árbol de herencia	2,17
Nivel de acoplamiento aferente	5,273
Nivel de acoplamiento eferente	2,182
Abstracción	0,244
Instanciabilidad	0,354
Distancia normalizada	0,427
Escasez de cohesión en métodos	0,425
Índice de especialización	0,033

Atendiendo a las métricas obtenidos observamos que la métrica 'Escasez de cohesión en métodos' tiene un valor de 0,427. Esta métrica debe tener un valor cercano a 0 por lo que se podría dividir en subclases aquellas clases que tengan una carga mayor de código. Lo mismo ocurre con 'Distancia normalizada'. Tiene que tener un valor cercano a 0, en este caso 0,427 por lo que se podría mejorar la estructuras de paquetes para una mejor organización del código.

2.3. Bibliotecas utilizadas

En este apartado se presentan aquellas bibliotecas externas que han sido utilizadas para la implementación de la aplicación:

- **WebLaf:** Biblioteca utilizada para mejorar el Look and Feel de Java y utilizar las distintas componentes gráficas que ofrece. La versión utilizada ha sido la 1.26. [7]

- **XStream:** Esta biblioteca se encarga de transformar un objeto Java a un formato a xml y de su recuperación. Ha sido utilizada para la escritura y recuperación de las distintas estructuras Java en un fichero xml. Esta biblioteca a su vez tiene dependencias de otras bibliotecas: hamcrest-core-1.3, xmlpull-1.1.3.1, jcommon-1.0.21 y xpp3_min-1.1.4c. La versión utilizada ha sido la 1.4.5.[8]

- **JFreeChart:** Esta biblioteca permite crear gráficos sobre estadísticas en Java. Fue utilizada para la construcción de los gráficos en la herramienta **Profiler**. La versión utilizada ha sido 1.0.27.[9]

- **Tools:** Biblioteca de la JDK (*Java Development Kit*) que permite la comunicación con la JDI, por lo que fue utilizada para la extracción de información sobre la ejecución de un programa Java.

- **Forms:** Biblioteca utilizada para crear layouts para los distintos formularios de la aplicación. La versión utilizada fue la 1.3.0.

- **TreeLayout:** Biblioteca utilizada para representar gráficamente árboles en la herramienta Inspector [10].

2.4. Estructura de la aplicación

Cabe mencionar que para producir una mayor independencia y modularidad, se ha utilizado el patrón de diseño MVP (*Model-View-Presenter*). Por ello a continuación se verá que todos los distintos módulos que contienen un módulo gráfico tienen a su vez un presentador y un modelo, produciendo así una independencia entre la lógica gráfica y la lógica de negocio de la aplicación. A continuación se describirán los cuatro módulos más importantes de la aplicación de una forma más detallada:

Paquete general

En este paquete como se comentó anteriormente se encuentra principalmente la funcionalidad de comunicar a los otros módulos (**Profiler**, **Tracer** e **Inspector**). Este paquete, además de servir de comunicador entre los distintos módulos, también contiene información o clases que son de utilidad para todos los paquetes, como pueden ser configuraciones, estructuras de datos, etc. El paquete general está compuesto de numerosos paquetes los cuales se describen a continuación:

- **com.general.model:** En este paquete se encuentra toda la lógica de negocio que es común para todos los módulos. En él se encuentran clases cuya funcionalidad se reduce a proporcionar una serie de métodos de ayuda para facilitar el desarrollo de la aplicación. Estas clases son **FileUtilities**, **XStreamUtil**, **ClassFinder** y **JarFinder**. Además también contiene estructuras de datos compartidas por varios módulos, como pueden ser **Data** o **MethodInfo**, contenidas dentro del paquete **com.general.model.data**.

    - **XStreamUtils**: Es una clase abstracta que contiene métodos útiles a la hora de trabajar con ficheros xml y con la biblioteca XStream, ya que contiene numerosos alias que se fueron predefinidos por el grupo a la hora de construir el fichero xml.

    - **ClassFinder**: Es una clase dedicada a la búsqueda recursiva a través de directorios de ficheros class para obtener el *classpath* (directorio donde se encuentran todos los class para la ejecución de un programa Java), para la búsqueda clases que contienen el método main para el arranque, etc.

    - **JarFinder**: Al igual que **ClassFinder** es una clase que sirve para la búsqueda recursiva a través de directorios de ficheros class, **JarFinder** sirve para encontrar ficheros jar ejecutables para la traza, o bien incorporar dichos ficheros jar como bibliotecas al *classpath*.

- **com.general.model.data:** En este paquete se encuentran todas aquellas estructuras de datos que se comparten entre los distintos módulos.

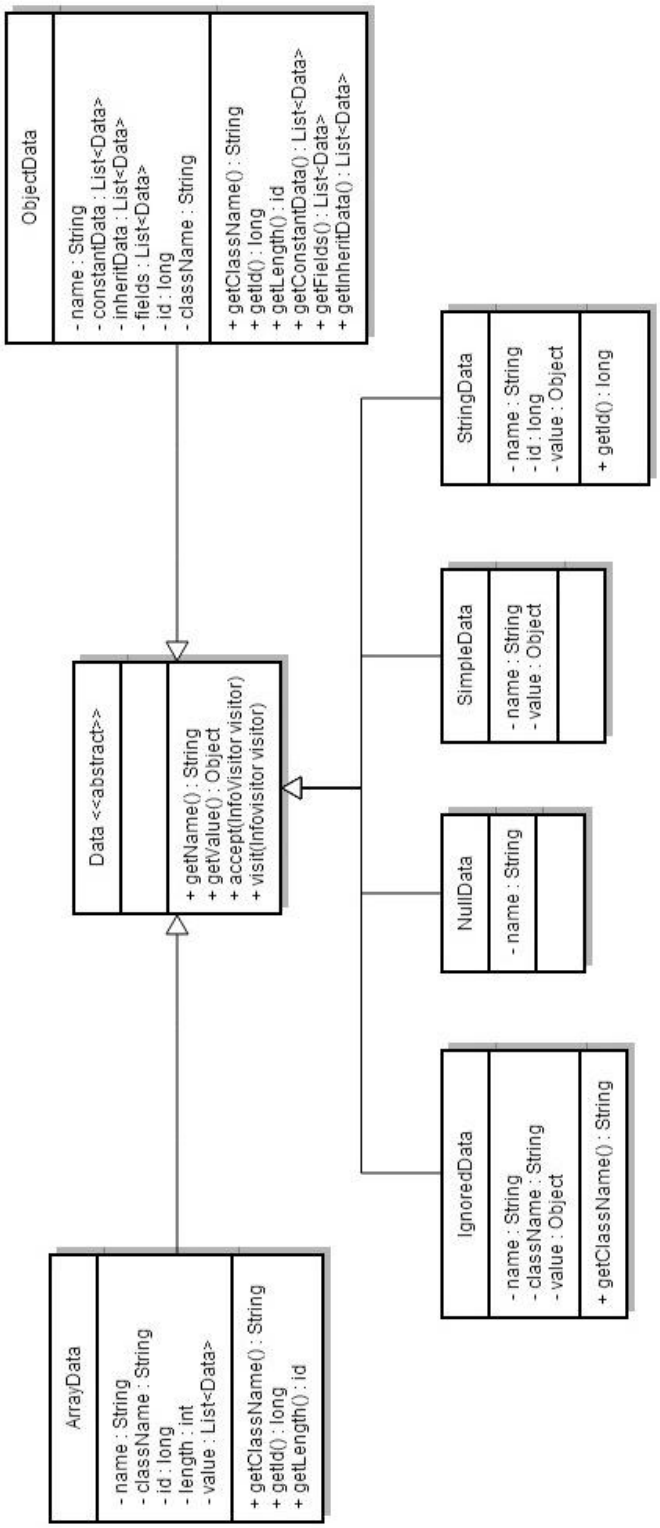


Figura 2.4: Diagrama UML del paquete com.model.data.variables

El paquete **com.general.model.data.variables** como se mostró en la figura 2.4, está formado por una clase abstracta **Data** simulando el comportamiento de una variable en Java. Cada clase que hereda **Data** es un tipo de datos que ofrece la JDI. Es importante mencionar que no se puede acceder directamente a los objetos de la máquina virtual, es decir, no se puede referenciar el objeto directamente sino que se debe acceder mediante una serie de métodos para sacar su id, nombre de variable, saber el tipo de la variable, etc. Debido a esto, se han utilizado las diversas clases de **Data** para construir el objeto que representa en su totalidad al objeto de la máquina virtual. Las clases contenidas en el paquete **com.general.model.data.variables** son:

- **ArrayData:** Utilizada para la representación de *Arrays*.
- **ObjectData:** Utilizada para la representación de objetos Java. Se diferencian entre atributos heredados, constantes y propios.
- **IgnoredData:** Utilizada para cuando se excluye una clase de la cuál se quiere omitir información. En este clase se guarda únicamente el nombre de la variable.
- **NullData:** Utilizada para la representación de valores *null*.
- **SimpleData:** Utilizada para la representación de los valores primitivos de Java (*byte*, *short*, *integer*, *long*, *float*, *double*, *char* y *boolean*)
- **StringData:** Utilizada para la representación de objetos *String*.

Además, el paquete **com.general.data** contiene tres clases adicionales, para representar otros tipos de información:

- **ChangeInfo:** Informa qué variable ha cambiado y qué valores ha tomado.
- **MethodInfo:** Representa la información de un método en la entrada y en la salida.
- **ThreadInfo:** Representa la información del *thread* actual.

- **com.general.presenter:** En este paquete se encuentra la clase **JavaTracerPresenter**, que hace de comunicador entre los distintos módulos de la aplicación.

- **com.general.resources:** En este paquete se encuentran todos los recursos gráficos como pueden ser imágenes, iconos y fuentes utilizadas acompañada por dos clases (**ImageLoader** y **FontLoader**) que facilitan la carga de los distintos de recursos.

- **com.general.settings:** La aplicación de **Settings** también utiliza el patrón de diseño MVP, por lo que tiene tres subpaquetes: **general.settings.presenter**, **general.settings.model** y **general.settings.view**. En estos paquetes están repartidas todas aquellas clases que permiten modificar y consultar la configuración de la aplicación. Las clases más importantes a destacar es la clase Settings la cual carga por defecto la configuración por defecto (almacenada en el fichero java-tracer-properties.xml) y la proporciona información sobre la configuración al resto de módulos.

- **com.general.view:** En este paquete se encuentran aquellas vistas o elementos gráficos que utilizan los distintos módulos de la aplicación:

    - **AboutDialog:** Dialog personalizado que muestra la versión y desarrolladores de la aplicación.

    - **WebFileChooserField:** Reimplementación de la clase *WebFileChooserField* utilizada por la biblioteca WebLaf. Esta componente es aquella que permite elegir el directorio donde se encuentran las clases a traza mediante un *FileChooser*.

    - **com.general.view.jtreetable:** Este paquete contiene todas las clases que permiten la implementación de una tabla con filas expandibles. En este paquete se encuentran las siguientes clases:

        - **JTreeTable:** Esta clase hereda *JTable* e implementa la lógica para expandir y contraer las filas de la tabla. Esta clase tiene asociado dos modelos, uno que simula la lógica de árbol (**TreeModel**) y otro que controla la lógica (*DefaultTableModel*).

        - **TreeModel:** Clase que implementa la estructura del árbol de la **JTreeTable.**

        - **TableTreeNode:** Clase que representa un nodo en la estructura de árbol implementada por **TreeModel**.

        - **TableRowData:** Interfaz usada por el **TreeModel** para generalizar la estructura del **TreeModel** y de esta forma conseguir introducir cualquier tipo de dato a la fila.

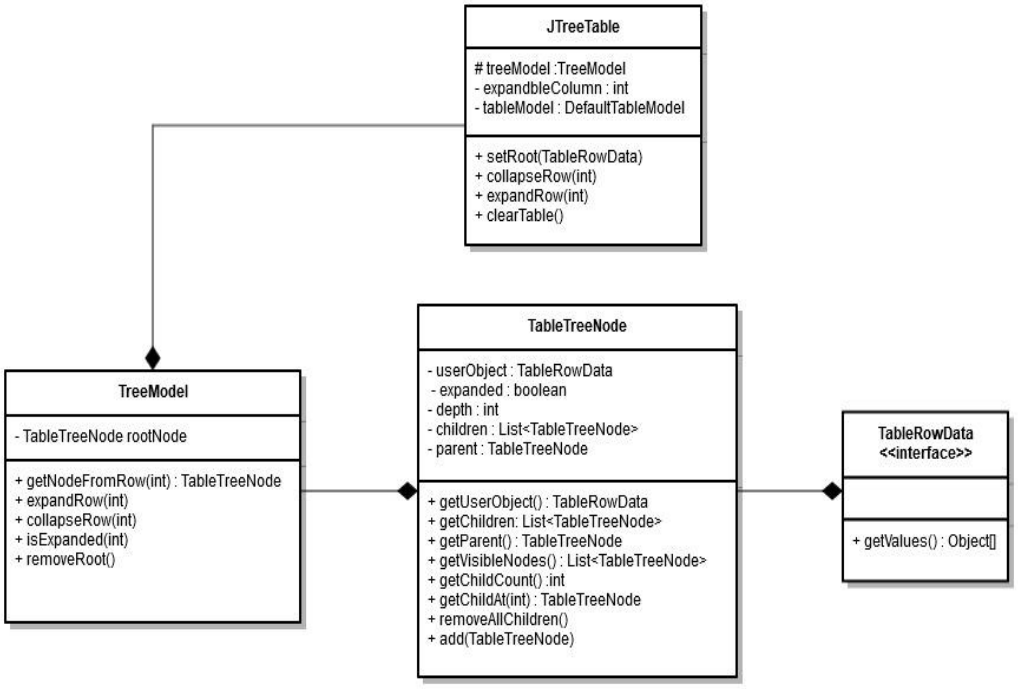


Figura 2.5: Diagrama UML del paquete com.general.view.jtreetable

Paquete tracer

Este paquete es el núcleo de la aplicación, en el se encuentra toda la lógica sobre cómo se traza el programa y las distintas opciones que hay a la hora de trazar. A continuación se describe en profundidad el paquete **com.tracer**:

- **com.tracer.arguments:** Este paquete está formado por **com.tracer.arguments.view** y **com.tracer.arguments.presenter**. En dichos paquetes se encuentran todas aquellas clases (siguiendo el patrón MVP) para la entrada de argumentos para la aplicación que será trazada a través de un interfaz gráfico.

- **com.tracer.console:** Este paquete, siguiendo el patrón de diseño MVP, está dividido en **com.tracer.console.model**, **com.tracer.console.view** y **com.tracer.console.presenter**. En todos ellos se encuentran aquellas clases relacionadas con la implementación de la consola. Las clases a destacar son las clases que se encuentran en:

- **StreamOutputRedirectThread:** Esta clase es un *thread* que tiene el objetivo de redirigir todo el flujo de salida por defecto de un programa al flujo de salida de la consola. Esto tiene la repercusión de que si se escribe en la salida estándar de un programa, se podrá observar dicha escritura en la consola de la aplicación.

- **StremInputRedirect:** Esta clase tiene como objetivo redirigir un flujo de entrada a la entrada estándar del programa a trazar. En este caso a diferencia del anterior no hace falta tener un thread leyendo constantemente el flujo, sino que en este caso cuando se termina de escribir la entrada esta se introduce directamente.

- **Console:** Esta clase junta todo aquello que tenga que ver con las entradas y salidas estándar. Esta clase posee dos **StreamOutputRedirectThreads** (uno para la salida estándar y otro para la salida de error) y un **StremInputRedirect** para la entrada por defecto. Con ello se tienen cubiertas todas las entradas y salidas por defecto de un programa Java.

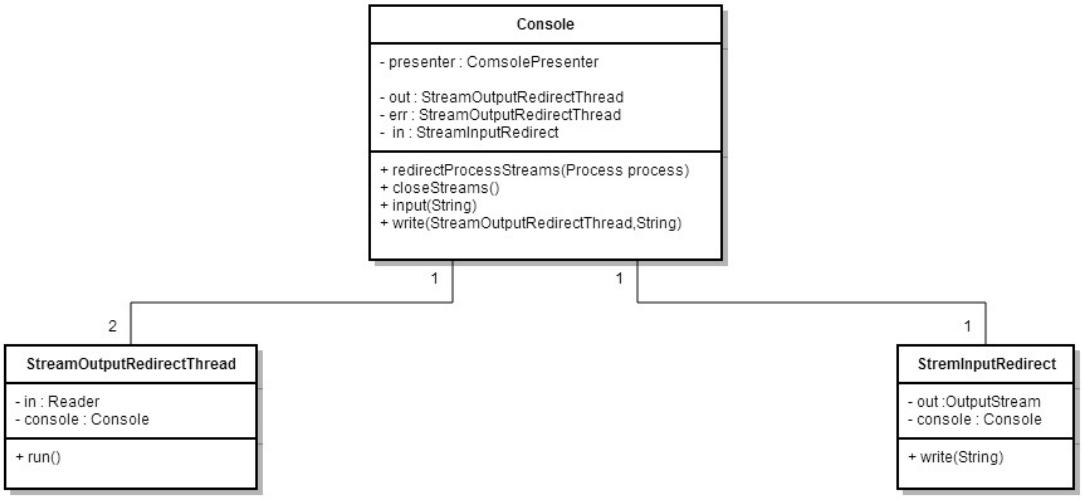


Figura 2.6: Diagrama UML del paquete com.tracer.console

- **com.tracer.model:** Este es uno de los paquetes más importantes de la aplicación. En él se encuentra el motor de arranque de la aplicación a depurar, el generador de eventos de depuración y el generador de la traza.

- **Tracer:** Esta clase es la encargada de realizar los procesos de lanzamiento de la aplicación e inicialización del motor de depuración. Para la inicialización de la aplicación a trazar se utiliza el conector *com.sun.jdi.CommandLineLaunch*, y se configuran las distintas opciones (añadido de bibliotecas externas, *classpath*, si es un archivo jar o no, etc) y la clase con el método *main* o jar desde donde empieza la aplicación. Cuando se ha lanzado la aplicación se crea un **EventThread** que sirve para la escucha de eventos sobre sucesos en la aplicación.

- **EventThread:** Esta clase se encarga de escuchar a los eventos que se producen a la hora de depurar la aplicación. En la arquitectura de la aplicación cada tipo de evento está controlado por un manager que realiza una acción en función del evento. Por ejemplo, cuando ocurre **MethodEntryEvent** el manager asociado recoge información del evento y lo graba en la traza. Los diferentes managers se encuentran en el paquete **com.tracer.model.managers**. Los eventos elegidos para elaborar la traza, mostrados en la figura 2.7, han sido:

    - **ExceptionEvent:** Ocurre cuando se ha producido una excepción en el programa. Se guarda la información del evento en la traza.
    - **MethodEntryEvent:** Ocurre cuando se llama a un método. Se guarda la información del evento en la traza.
    - **MethodExitEvent:** Ocurre cuando el método va a finalizar su ejecución. Se guarda la información del evento en la traza.
    - **ThreadDeathEvent:** Ocurre cuando termina la ejecución de un *thread* asociado a la aplicación. Utilizado para finalizar la traza.
    - **VMDeathEvent:** Ocurre cuando la virtual machine para la ejecución del proceso. Utilizado para finalizar la traza.
    - **VMDeathDisconnect:** Ocurre cuando se pierde la conexión con la máquina virtual. Utilizado para finalizar la traza en caso de que se haya perdido conexión con la aplicación (cierre forzoso).

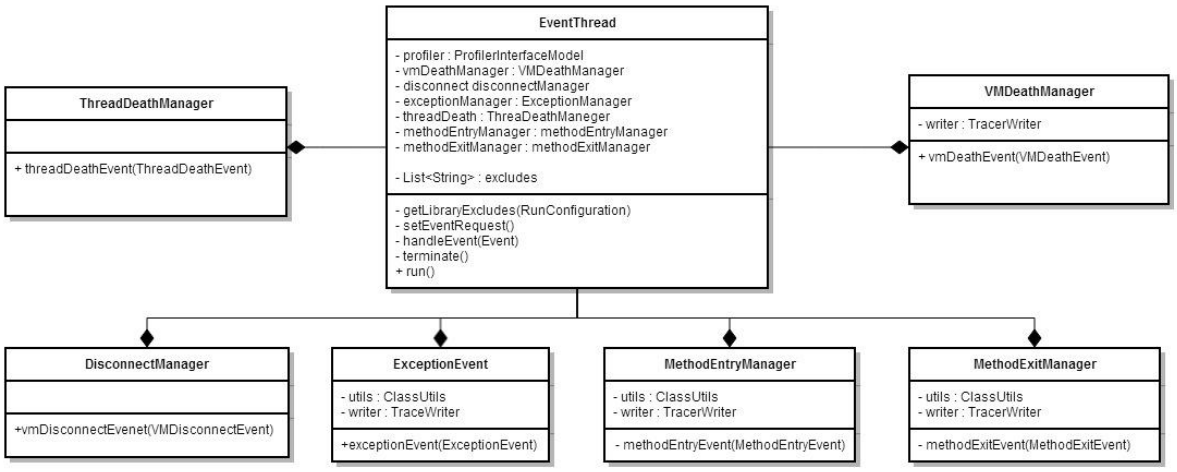


Figura 2.7: Diagrama UML de la clase EventThread

- **TracerWriter:** Se encarga de escribir la información de métodos como **MethodEntryInfo** y **MethodExitInfo** en el archivo xml.

- **ChangeDetector:** Esta clase se utiliza para comparar dos objetos y ver qué atributos han cambiado.

- **ClassUtils:** Esta clase es la encargada de generar toda aquella información a través de la JDI proporcionada por Java transformando dichos objetos a una estructura de datos **Data**.

- **RunConfiguration:** Esta clase almacena todos aquellos datos relativos con la configuración del lanzamiento de la aplicación como pueden ser la clase con un método *main*, si es jar o no, bibliotecas externas, *classpath*, etc.

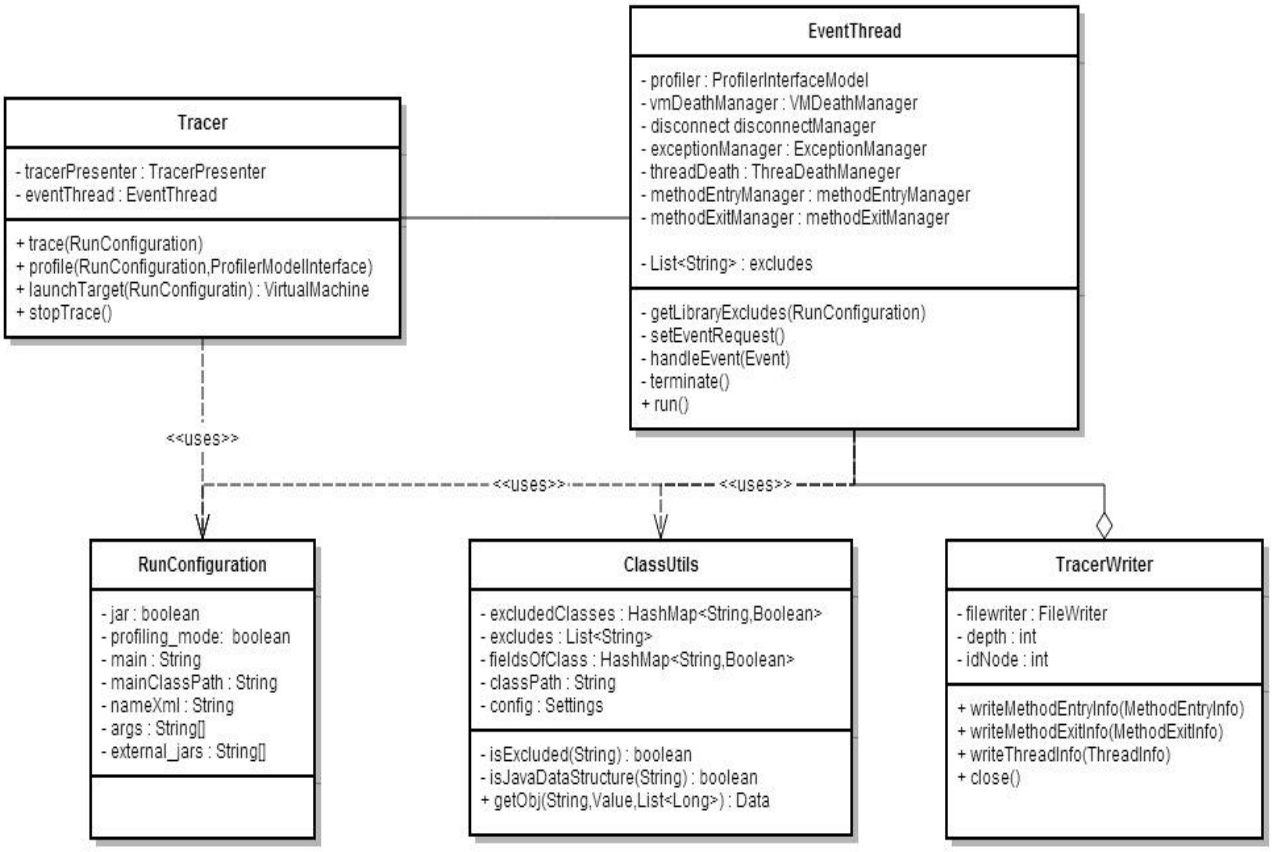


Figura 2.8: Diagrama UML sobre el paquete com.tracer.model

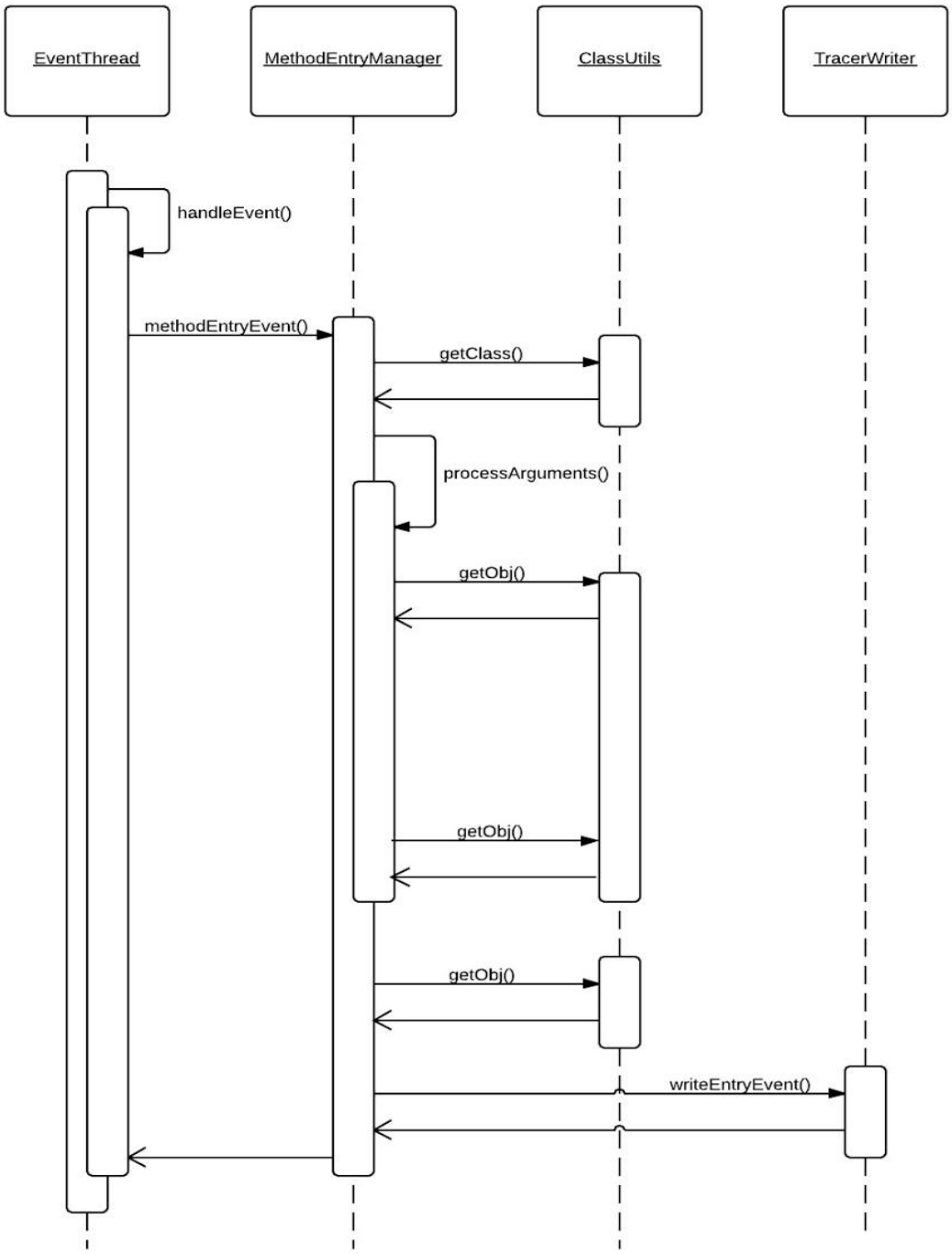


Figura 2.9: Diagrama de secuencia para registrar la llamada a un método.

- **com.tracer.presenter:** En este paquete se encuentra el presentador del tracer **TracerPresenter** que comunica el modelo (**com.tracer.model**) con la vista (**com.tracer.view**) además de los subpaquetes de la entrada de argumentos (**com.tracer.arguments**) y la consola (**com.tracer.console**). A su vez se comunica con el presentador general de la aplicación (**JavaTracerPresenter**).

- **com.tracer.view:** Este paquete almacena todo aquello que tiene que ver con la vista del **Tracer**.

Paquete profiler

En este paquete se encuentra todo aquello relacionado con el profiler, la forma de recoger información, cómo mostrarla, cómo sacar las estadísticas de dicha información, etc. En este paquete se sigue aplicando el patrón de diseño MVP, por lo que se distinguen en tres subpaquetes:

- **com.profiler.model:** Este paquete recoge toda aquella lógica relacionada con la tarea de hacer un *profile* de la aplicación a trazar, con el objetivo de ver una serie de estadísticas sobre la aplicación. De este paquete se destaca:

    - **Profiler:** Se encarga de registrar cuando hay un evento de tipo *MethodEntryEvent* en **EventThread**, es decir, registrar el método al que se ha llamado. Si ya estaba registrado, se aumenta el número de llamadas con lo que al final de la ejecución del programa se obtienen el número de llamadas de todos los métodos. El registro de los métodos se hace mediante una estructura de árbol (**ProfilerTree**) que se describe a continuación.

    - **ProfilerTree:** Es el árbol donde se añaden y se registran los métodos por **Profiler**. La estructura trata de guardar de forma jerárquica los distintos paquete, clases y métodos. Esta forma de jerarquía contiene la siguiente especificación:

        - **Paquetes:** Puede almacenar tanto clases como paquetes.
        - **Clases:** Solo pueden almacenar métodos.
        - **Métodos:** Son los nodos hoja del árbol.

Esta jerarquía se usa para que al registrar un método, aumente el número de llamadas de los padres (es decir, la clase a la que pertenecía el método) y los distintos paquetes a los que pertenece dicha clase. Como resultado, tendremos el número de llamadas que han ocurrido en una clase y dentro de un paquete. La siguiente figura describe gráficamente cómo debe anotar el **ProfilerTree** los métodos que van ocurriendo a lo largo del profiling.

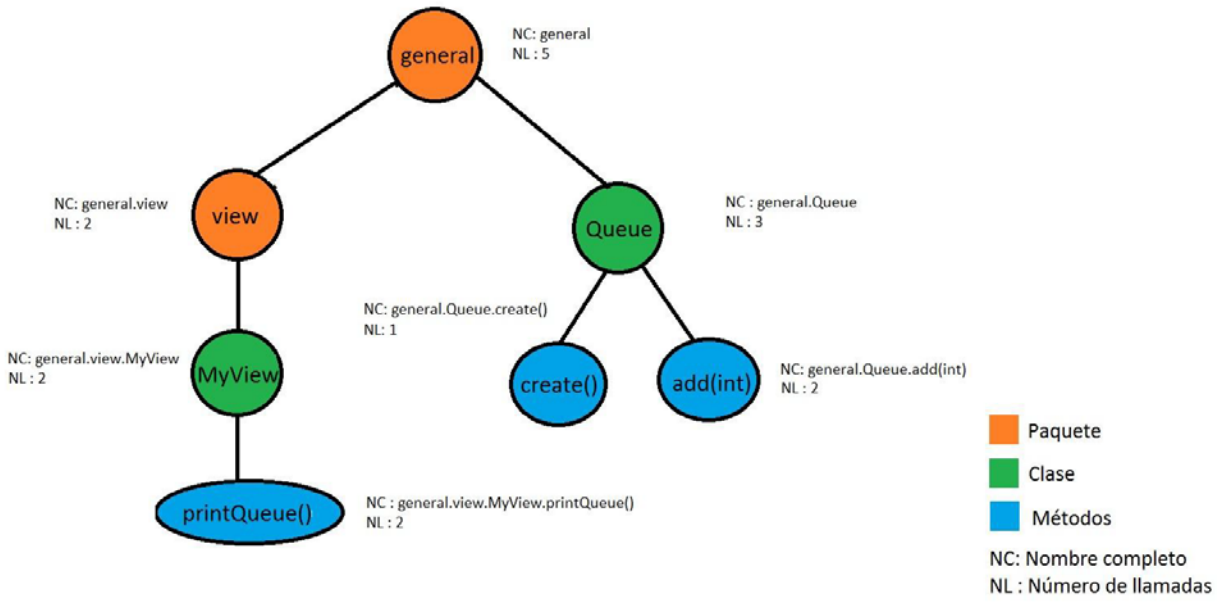


Figura 2.10: Ejemplo de la estructura de un ProfileTree

- **com.profiler.model.data:** En este paquete se encuentran distintas clases para la construcción del **ProfileTree**. Las clases más importantes son:

  - **ProfileData:** Es una clase abstracta que define el tipo de nodo del árbol e implementa las operaciones básicas de los árboles (*remove, add, getChildren, getParent*, etc.). También administra las funciones que son comunes a los distintos tipos de clases que heredan de ella como la suma del número de llamadas.

  - **ProfilePackage:** Esta clase hereda **ProfileData** y representa un paquete dentro de **ProfileTree**. Contiene información sobre el nombre del paquete o subpaquete y su nombre completo para poder identificarlo unívocamente.

- **ProfileClass**: Esta clase hereda **ProfilerData** y representa un nodo clase dentro de **ProfileTree**. Contienen información sobre el nombre de la clase abreviado y el nombre completo de la clase para poder identificarlo unívocamente.

- **ProfileMethod**: Esta clase hereda **ProfilerData** y representa un nodo método dentro de **ProfileTree**. Contienen información sobre el nombre del método y el nombre completo (nombre completo de la clase más el nombre del método) para poder identificarlo unívocamente.

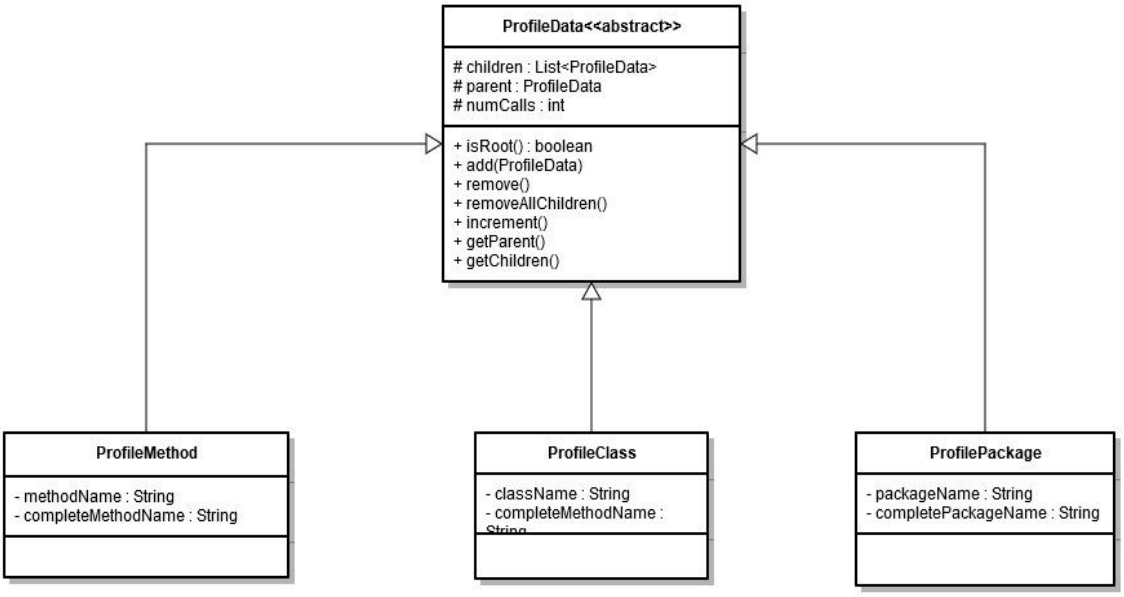


Figura 2.11: Diagrama UML del com.profiler.mode.data

- **com.profiler.presenter**: En este paquete se encuentra **ProfilePresenter** que es el encargado de comunicar el modelo y la vista. Aparte también se comunica con **JavaTracerPresenter** para comunicarse con el resto de módulos.

- **com.profiler.view**: En este paquete se encuentra el módulo gráfico que permite mostrar los gráficos y estadísticas del profile de una aplicación. Se encuentra en este paquete dos tipos de clases, unas destinadas a la tabla expandible y otras a la muestra de gráficos en forma de rosco utilizando la librería JFreeChart.

Paquete inspector

En este paquete se muestra el flujo del programa de manera sintetizada con una estructura de árbol, mostrando todas sus llamadas, representadas en forma de nodos del mismo, y para más información se tiene una tabla expandible, donde se muestran datos que corresponden al nodo que está seleccionado.

- **com.inspector.model:** Se parte sabiendo que antes de poder visualizar el árbol en el **Inspector**, la aplicación genera una traza en un archivo xml, que va a contener toda la información necesaria para poder pintar el árbol. Para la construcción de dicho xml, se utiliza la biblioteca XStream. XStream es una librería Java que permite serializar objetos Java a xml y realizar su posterior recuperación. En este paquete se encuentra la clase **XmlManager**, que contiene las consultas de este xml necesarias tanto para pintar el árbol en forma gráfica, como para la tabla expandible, la lógica de las consultas se encuentra en este paquete. Para dichas consultas se utiliza la biblioteca XPath de java.

- **com.inspector.objectinspector: ObjectInspector** es una de las dos partes que constituyen el **Inspector**. Esta hace referencia a la tabla que se muestra en la parte derecha, que contiene el *this* y el nodo seleccionado (método), junto con sus parámetros de entrada y el valor que devuelve. **ObjectInspector** consta a su vez de su modelo, presentador y vista.

    - **com.inspector.objectinspector.model:** En este caso el modelo está representado por tres clases, todas ellas implementadas con el patrón de diseño visitor. Este patrón es el más adecuado cuando se tienen que reconocer **ArrayData, StringData, NullData, IgnoredData, ObjectData, SimpleData** (explicados en el paquete general). Este paquete está constituído por toda la lógica que corresponde a mostrar la información de los tipos nombrados anteriormente citados, debido a que dependiendo del tipo de cambios producidos, la información se trata de manera diferente.

    - **com.inspector.objectinspector.presenter:** En este paquete se tiene **ObjectInspectorPresenter**, que como todo presentador es quién se encarga de comunicar el modelo y la vista. Pero a su vez también se comunica con **InspectorPresenter**, que es el encargado de comunicar tanto a la parte de **ObjectInspector**, como el **TreeInspector**, para que así puedan funcionar de una manera sincronizada.

    - **com.inspector.objectinspector.view:** En este paquete se encuentra el motor gráfico, que está constituído por una tabla expandible, en donde se muestra la información que corresponde al nodo actual del árbol seleccionado (**TreeInspector**).

- **com.inspector.treeinspector: TreeInspector,** es la otra parte por la que está constituído el **inspector**. Aquí es donde se muestra en forma de árbol todo el flujo del programa que se está tratando.
    - **com.inspector.treeinspector.presenter:** En este paquete se tiene a **TreeInspectorPresenter**, que al igual que en el **ObjectInspector**, a parte de comunicar su modelo con su vista, también comunica a éste con el **Objectinpector**.
    - **com.inspector.treeinspector.view:** Este paquete abarca toda la parte gráfica correspondiente al árbol que se muestra del flujo del programa. Haciendo uso de una biblioteca llamada TreeLayout, que permite pintar el árbol de forma gráfica, pudiendo hacer modificaciones, para adaptarlos a los propósitos planteados. Se implementa la forma de cómo se muestran los nodos del árbol.

- **com.inspector.presenter:** En este paquete se encuentra **InspectorPresenter**, que es el encargado de comunicar la vista, con el modelo, pero a su vez también, contiene JavaTracerPresenter, es decir para poder comunicarse con el flujo principal de la aplicación.

- **com.inspector.view:** Es todo correspondiente a la vista general del inspector, es decir las dos vistas unidas la del **ObjectInspector** con la **Treeinspector**, formando así aparentemente una vista, aunque por debajo estén catalogadas como diferentes, en este paquete se encarga de comunicarlas entre sí.

2.5. Dificultades encontradas

A lo largo del desarrollo del proyecto, se han tenido que afrontar problemas que suponen un riesgo a la hora de crear una aplicación realmente práctica. No obstante, se supo afrontar estos problemas mediante soluciones acertadas y cumpliendo el objetivo de construir una aplicación útil y práctica para el usuario.

Tamaño de la traza

Uno de los mayores problemas fue al excesivo tamaño de las trazas. El tamaño de los ficheros de traza depende fundamentalmente del número de llamadas y del tamaño de los objetos en el momento que se llama a los métodos. En una aplicación con unas 100.000 llamadas puede crear fácilmente ficheros de 300 MB. Esto a la hora de leer un xml es un problema, y se conoce como el problema de los paréntesis abiertos y cerrados. Este problema representa la necesidad de saber que con que parentesis cerrado se cierra uno abierto. El lenguaje de marcas xml está formado por etiquetas de apertura y etiquetas de cierre. Sin embargo, para saber a qué elemento se está refiriendo es necesario tener almacenado todas las etiquetas anteriores (incluido el contenido). Esto supone un problema ya que conforme van aumentando el número de paréntesis abierto aumenta el gasto en memoria. Por lo que las hay grandes problemas en cuanto al rendimiento con las consultas XPath en trazas que sean superiores a 100 MB.

La solución tomada fue añadir una serie de mejoras a la aplicación permitiendo ser posible el uso de ficheros xml en trazas grandes y soluciones para reducir el tamaño del fichero. Hubo una optimización de las consultas de los ficheros a la hora de coger los aquellos métodos que fueron llamados desde un método. En vez de coger el número de hijos (esta era la consulta que más tardaba), y recoger los hijos en un bucle con dicho de hijos (otra consulta), se optó por coger directamente el hijo k-ésimo. De esta manera, cuando se tiene un número N de hijos y la consulta intentaba acceder al N+1 el tiempo de error era menor comparada con la consulta de coger el número de hijos. Cuando daba dicha excepción se sabía que había terminado de recorrer los hijos. Esto acortó bastante el tiempo de carga en el inspector, lo cuál hacía procesables trazas mayores (hasta los 300 MB) que es el límite que daba Java para tratar con ficheros xml.

Sin embargo, estos 300 MB contenían información irrelevante para el proceso de depuración. Para paliar este problema se crearon de una serie de mejoras al trazador para que pudiese excluir información que el usuario no quisiese, haciendo así un árbol más limpio y a la vez reduciendo el tamaño de la traza, lo que permite a Java leer el fichero sin que el programa colapse el heap. Estas opciones se enumeran y explican a continuación:

- **Excluir paquetes, clases o métodos:** Sin duda la mejora que más mejora la traza. Esta opción permite al usuario añadir información que quiere excluir, impidiendo que se grabe en la traza. De esta forma, si el usuario es capaz de elegir bien la información que se quiere grabar, se consigue reducir de forma drástica el tamaño del fichero.

- **Excluir this:** En Java casi siempre que se llama a un método es llamado desde un objeto que contiene una cierta información. A este objeto, se puede acceder a través de la expresión *this*. No obstante, hay en casos excepcionales en los que únicamente se quiere ver qué cambios han ocurrido en los argumentos de los métodos. Por ello, se añadió esta opción que además ayuda a la reducción del fichero de traza.

- **Excluir Java Data Structures:** Por defecto se excluyen los paquetes java.*, javax.*, sun.* y com.sun.*, porque se grabará continuamente información interna de Java que no interesa al usuario. No obstante, al excluir java.* se está a su vez excluyendo java.util.*, donde se encuentran las estructuras de datos como *ArrayList*, *HashMap*, *Set*, *Vector*, etc. Esta opción permite excluir este tipo de estructuras para reducir el tamaño de la traza.

- **Excluir bibliotecas externas:** Esta opción es interesante porque normalmente los programas Java se basan en funcionalidad aportada por bibliotecas externas. Casi siempre, cuando el usuario va a depurar una aplicación ignora, las bibliotecas bien porque da por hecho que funcionan o bien porque no son accesibles. JavaTracer da la opción de añadir o no a la traza o no las bibliotecas que se requieren para el funcionamiento del programa. Si se decide excluirla se reduce bastante el peso del fichero de traza.

En resumen, el programa está limitado por el tamaño del fichero de traza a la hora de ser procesado por el **Inspector** (aproximadamente 300 MB). No obstante hay numerosas medidas para permitir al usuario omitir información que no es relevante para la depuración, facilitando la depuración y creando ficheros menores.

Representación del árbol:

Otra de las dificultades fue representar los árboles gráficamente para poder mostrar árboles de métodos, así como la posibilidad de plegar y desplegar los nodos, permitiendo al usuario explorar el árbol de llamadas de forma rápida e intuitiva.

La primera solución fue la utilización de la biblioteca, **yEd**. Permitía desplegar y plegar nodos y representar el árbol de distintas maneras, por lo que se encontró aquello que se necesitaba. No obstante, después de probarla e integrarla hubo un problema, la licencia era de 30 días, por lo que se tuvo que buscar otra solución [11].

No se encontró ninguna biblioteca más que proporcionase las mismas prestaciones que **yEd**. Se optó por buscar una biblioteca capaz de representar los árboles de la manera que se quería, pero sin ninguna funcionalidad más y con la capacidad de modificar el código para poder personalizar dicho árbol. Se encontró la biblioteca TreeLayout, una biblioteca de software libre que permitía representar el árbol de forma gráfica, pero a la vez permitía poder modificarla para adaptarla a los requisitos buscados. Tras una serie de modificaciones se consiguió añadir toda la funcionalidad que se requería y personalizarla en cuanto a aspecto visual.

Entrada y salida de un programa

Una de las tareas pendientes era mostrar la entrada y salida de error de un programa por defecto. No había manera de que los programas que dependían de una entrada por consola funcionasen en JavaTracer. Tampoco se podían ver todas aquellas salidas de error y mensajes en la salida principal que muestre dicho programa.

La primera solución propuesta fue la de redirigir la entrada y salida a ficheros, de manera que, la entrada leería línea a línea del fichero, produciendo así un flujo de entrada de datos y las salidas y errores a los ficheros en los cuales se escribiría todo aquello que produjese el flujo de salida de datos. Esta solución fue descartada por problemas en la entrada, ya que muchas veces la entrada puede ser variable dependiendo de la aleatoriedad del programa, por lo que se optó por hacer una consola a través de la cual, se puede interactuar con el programa.

La entrada/salida de texto en la aplicación se basó en crear una consola parecida a la de Eclipse. En ella se puede escribir aquello que se desee introducir como entrada y a la vez en la misma consola se podrán ver las salidas y los errores que se muestran normalmente en un programa permitiendo al usuario una comodidad buena a la hora de tener que interactuar con el programa.

2.6. Etapas del proyecto

Al comienzo del proyecto, se eligió un desarrollo en espiral. Se optó por una metodología en espiral para ir pudiendo ver el avance gradual de la aplicación de una reunión con respecto a la siguiente. Con esta metodología se podía corregir las carencias o los fallos de la aplicación para la siguiente reunión y a la vez tener constantemente un feedback por parte del director del proyecto. Las reuniones se desarrollaban con carácter semanal a excepción de dos periodos en los cuales son del 19/12/2014 al 09/01/2014 (vacaciones de Navidad) y del 16/01/2014 al 20/02/2014 (periodo de exámenes).

En las figura 10 y 11 se mostrará un diagrama Gantt de la evolución del proyecto. En dichas figuras se podrá observar todos aquellos cambios importantes realizados sobre la aplicación y la importancia de la metodología en espiral en nuestro proyecto.

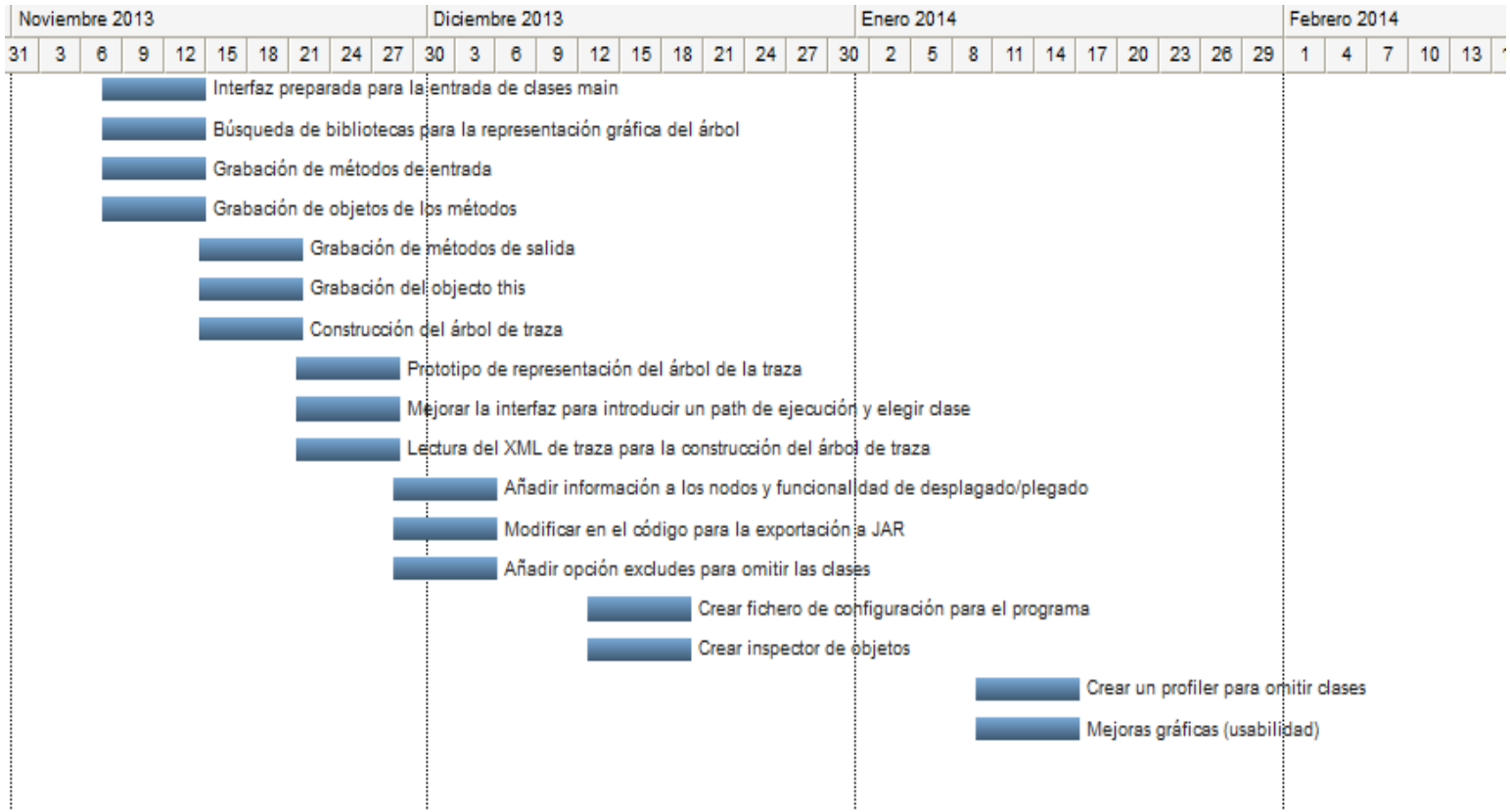
En la siguientes figuras 10 y 11 se muestran aquellas modificaciones más importantes realizadas en el programa con una fecha de inicio (día de la reunión) y una fecha de fin (día de la siguiente reunión). Las fechas comprendidas entre las dos anteriores corresponden a los días de desarrollo (6 días).

Finalmente en el diagrama de Gantt se podrá ver las tareas realizadas de una forma más gráfica. También se podrán sacar conclusiones de la continuidad de las reuniones, el número de tareas importantes realizadas de una semana a otra, etc.

Nombre	Duración	Inicio	Fin
Interfaz preparada para la entrada de clases main	6d	07/11/2013	14/11/2013
Búsqueda de bibliotecas para la representación gráfica del árbol	6d	07/11/2013	14/11/2013
Grabación de métodos de entrada	6d	07/11/2013	14/11/2013
Grabación de objetos de los métodos	6d	07/11/2013	14/11/2013
Grabación de métodos de salida	6d	14/11/2013	21/11/2013
Grabación del objecto this	6d	14/11/2013	21/11/2013
Construcción del árbol de traza	6d	14/11/2013	21/11/2013
Prototipo de representación del árbol de la traza	6d	21/11/2013	28/11/2013
Mejorar la interfaz para introducir un path de ejecución y elegir clase	6d	21/11/2013	28/11/2013
Lectura del XML de traza para la construcción del árbol de traza	6d	21/11/2013	28/11/2013
Añadir información a los nodos y funcionalidad de desplagado/pleg	6d	28/11/2013	05/12/2013
Modificar en el código para la exportación a JAR	6d	28/11/2013	05/12/2013
Añadir opción excludes para omitir las clases	6d	28/11/2013	05/12/2013
Crear fichero de configuración para el programa	6d	12/12/2013	19/12/2013
Crear inspector de objetos	6d	12/12/2013	19/12/2013
Crear un profiler para omitir clases	6d	09/01/2014	16/01/2014
Mejoras gráficas (usabilidad)	6d	09/01/2014	16/01/2014
Añadir excepciones al árbol	6d	20/02/2014	27/02/2014
Añadir la posibilidad de meter argumentos a la aplicación	6d	20/02/2014	27/02/2014
Comunicación entre las aplicaciones	6d	20/02/2014	27/02/2014
Generaciónde una aplicación profiler	6d	20/02/2014	27/02/2014
Mejoras de usabilidiad para el usuario	6d	20/02/2014	27/02/2014
Mejoras de usabilidiad para el usuario	6d	20/02/2014	27/02/2014
Comunicación entre las aplicaciones	6d	27/02/2014	06/03/2014
Mejoras gráficas y de usabilidad para la aplicación profiler	6d	27/02/2014	06/03/2014
Creación de la consola	6d	06/03/2014	13/03/2014
Creación de una interfaz para la configuración de la aplicación	6d	06/03/2014	13/03/2014
Añadir estadísticas al profiler	6d	06/03/2014	13/03/2014

Mejora de la navegación entre las aplicaciones	6d	06/03/2014	13/03/2014
Mejoras en la interfaz general y usabilidad en el profiler	6d	13/03/2014	20/03/2014
Mejora de la consola	6d	13/03/2014	20/03/2014
Arreglar problemas con trazas grandes (300 MB)	6d	13/03/2014	20/03/2014
Añadir opción de excluir métodos	6d	13/03/2014	20/03/2014
Generación automática de ficheros traza y profiler	6d	20/03/2014	27/03/2014
Mejorada la usuabilidad de aplicaciones JAR	6d	20/03/2014	27/03/2014
Funcionamiento multiplataforma	6d	20/03/2014	27/03/2014
Añadir la posibilidad de meter argumentos a la aplicación	6d	20/03/2014	27/03/2014
Añadidas mejoras e incluidas en la configuración de la aplicación	6d	20/03/2014	27/03/2014
Mejoras gráficas y creación de hilos para aumentar rendimiento	6d	27/03/2014	03/04/2014
Retoques finales	6d	03/04/2014	10/04/2014

Figura 2.12: Listado de tareas del diagrama Gantt


Noviembre 2013
Diciembre 2013
Enero 2014
Febrero 2014
31 3 6 9 12 15 18 21 24 27 30 3 6 9 12 15 18 21 24 27 30 2 5 8 11 14 17 20 23 26 29 1 4 7 10 13
Interfaz preparada para la entrada de clases main
Búsqueda de bibliotecas para la representación gráfica del árbol
Grabación de métodos de entrada
Grabación de objetos de los métodos
Grabación de métodos de salida
Grabación del objecto this
Construcción del árbol de traza
Prototipo de representación del árbol de la traza
Mejorar la interfaz para introducir un path de ejecución y elegir clase
Lectura del XML de traza para la construcción del árbol de traza
Añadir información a los nodos y funcionalidad de desplagado/plegado
Modificar en el código para la exportación a JAR
Añadir opción excludes para omitir las clases
Crear fichero de configuración para el programa
Crear inspector de objetos
Crear un profiler para omitir clases
Mejoras gráficas (usabilidad)

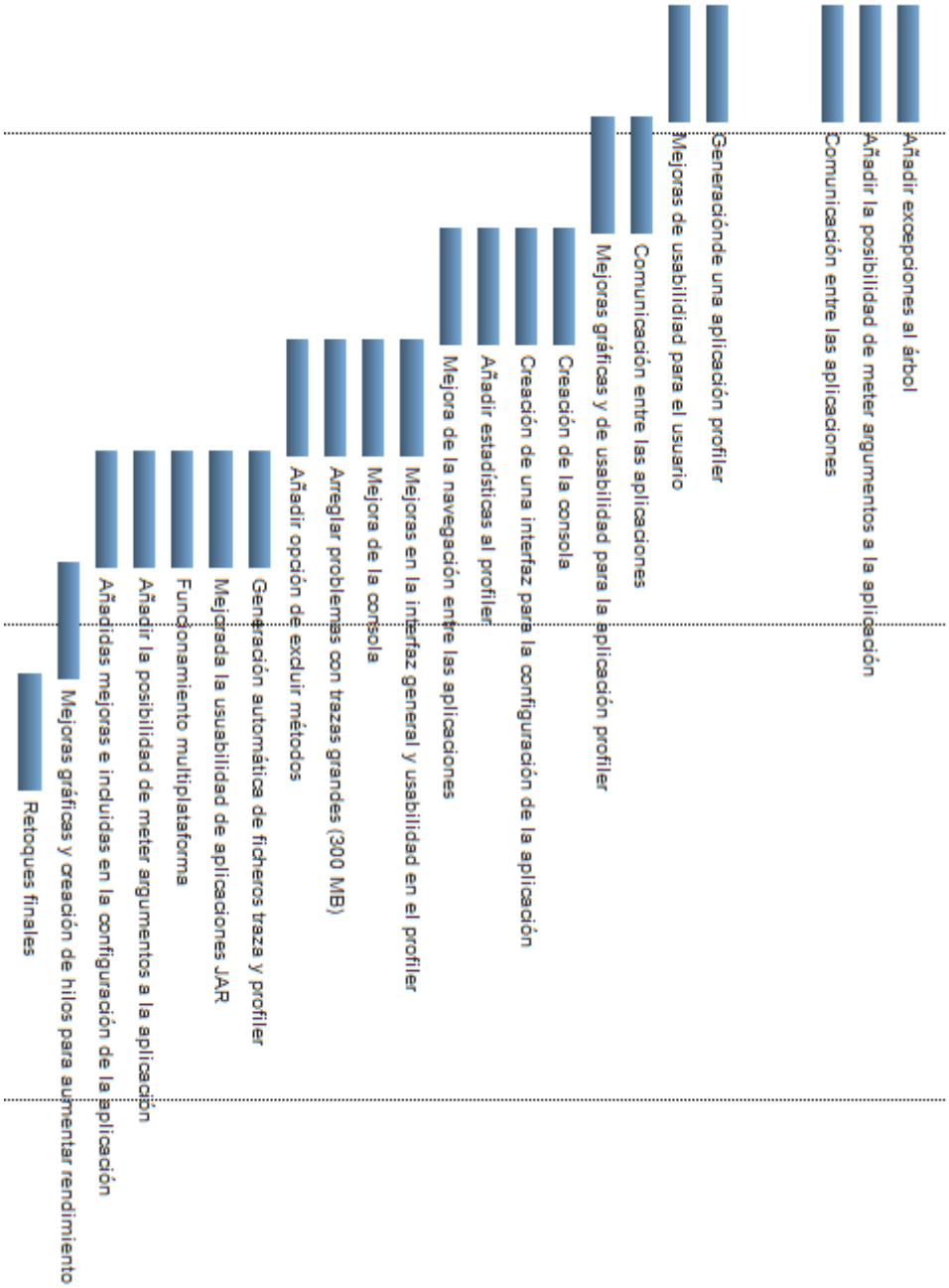


Figura 2.13: Diagrama Gantt del proyecto

2.7. Conocimientos empleados

Esta aplicación no hubiese sido posible sin una formación base previa adquirida a lo largo de la carrera. En este apartado se recoge todo aquello que se ha visto a la carrera y se ha aplicado en el proyecto demostrando la relevancia de las asignaturas en nuestro proyecto.

En el aspecto de la programación han influido numerosas asignaturas a la hora de desarrollar el proyecto. Desde el ámbito de la programación imperativa como Introducción a la Programación, Programación Orientada a Objetos o Laboratorio de Programación (1, 2 y 3) han sido esenciales para tener una noción amplia de cómo programar en distintos lenguajes. Entre todas ellas se destaca Laboratorio de Programación 3 y Programación Orientada a Objetos porque ambas se dan desde el contexto de Java, lenguaje que se ha utilizado para el desarrollo del proyecto.

No obstante también han sido útiles a la hora de programar de forma modular y ordenada las asignaturas de Programación Orientada a Objetos o Ingeniería del Software donde se aprenden numerosos patrones que se han aplicado como pueden ser Observador/Observable, el patrón de Visitor o el patrón MVP (Model-View-Presenter). Gracias a todos los patrones, metodologías de desarrollo y conocimiento en modularizar ha sido posible un desarrollo cómodo de la aplicación.

También asignaturas como Metodología y Tecnología de la programación, Programación Lógica y Programación Funcional han sido bastante útiles ya que sirven bastante a la hora de pensar en recursividad. La recursividad en JavaTracer ha sido fundamental y sin estos conocimientos en dichas asignaturas hubiese sido imposible el desarrollo.

La asignatura de Estructuras de Datos y de la Información fue de gran ayuda al tener que representar y tratar con numerosas estructuras de datos de forma trivial. En la aplicación se utilizan numerosas estructuras de datos como pueden ser listas, árboles, tablas hash, etc. y en dicha asignatura se aprende todo lo relacionado a dichas estructuras de datos haciendo que la aplicación fuese lo mejor posible.

A la hora de tratar con ficheros de datos ha sido útil la asignatura de Bases de Datos y Sistemas de la Información ya que en dicha asignaturas se aprende todo aquello sobre xml (el formato de ficheros que se utiliza) y consultas a través de XQuery y XPath, las cuales fueron de gran ayuda a la hora de leer el fichero xml y recoger la información de una forma óptima.

Otro aspecto en el que han influido las asignaturas ha sido en la memoria sin ir más lejos. Todos los detalles técnicos de la aplicación no hubiese sido posible sin una formación previa en la asignatura de Ingeniería del Software. En esta asignatura se aprendió todo aquellos aspectos fundamentales del desarrollo de un proyecto software como el diseño de un UML (Unified Modeling Language), diagramas de Gantt para tener una información visual de la planificación y del progreso del proyecto.

Unos conocimiento básicos de Sistemas Operativos y Laboratorios de Sistemas Operativos permitieron hacer de forma rápida JavaTracer multiplataforma. También se usaron conocimientos adquiridos en estas asignaturas como cerrojos, *multithreading*, descriptores de fichero, redirecciones, etc. que permitieron mejorar la aplicación de forma significativa.

2.8. Bibliografía

[1] *Java Platform Debugger Architecture*: Esta descripción está tomada de http://docs.oracle.com/javase/7/docs/technotes/guides/jpda
[2] *Java Debugger Interface*: Esta descripción está tomada de http://docs.oracle.com/javase/7/docs/technotes/guides/jpda /architecture.html#jdi
[3] Metrics: http://metrics.sourceforge.net/
[4] Eclipse : https://www.eclipse.org/
[5] *Tracer*: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/trace.html
[6] Índice de especialización por clase: http://www.ssmmr.org/precalculos/fuentes/ayuda/ES/fuentes/metricas/iepclase.html
[7] Biblioteca *WebLaf* : http://weblookandfeel.com/
[8] Biblioteca XStream: http://xstream.codehaus.org/
[9] Biblioteca JFreeChart: http://www.jfree.org/jfreechart/
[10] Biblioteca TreeLayout: https://code.google.com/p/treelayout/
[11] Biblioteca yEd: http://www.yworks.com/en/products_yed_about.html
[12] Biblioteca Gantt : http://es.wikipedia.org/wiki/Diagrama_de_Gantt

3. Funcionamiento de la aplicación

En este capítulo se mostrará todo lo relacionado con el uso de la herramienta. Las secciones de este capítulo son:

1. **Funcionamiento:** Información acerca de cómo usar la herramienta, qué opciones y menús tiene y qué resultados se obtienen, incluyendo un ejemplo de uso mediante un programa que implementa el algoritmo de ordenación *MergeSort*.

2. **Limitaciones:** Se exponen aquellas causas por las cuales la herramienta tiene capacidades limitadas.

3. **Casos de estudio:** Se citan una serie de casos de prueba los cuales han sido usados para probar la aplicación.

3.1. Funcionamiento

Para utilizar la aplicación JavaTracer se necesita tener instalado la versión 1.7 o superior de Java por lo que se necesitarán uno de los siguientes productos:

- **JRE o *Java Runtime Environment*:** El conjunto de aplicaciones y bibliotecas necesarias para utilizar una aplicación Java.

- **JDK o *Java Development Kit*:** Esta formado por JRE y el kit para poder desarrollar aplicaciones en este lenguaje de programación.

La aplicación JavaTracer puede descargarse desde el siguiente enlace: http://sourceforge.net/projects/javatracer/files/?source=navbar.

La aplicación funciona en Windows, Linux y Mac OS X, la única diferencia es la manera de ejecutarla.

Una vez que se arranca la aplicación aparece la siguiente pantalla de inicio:

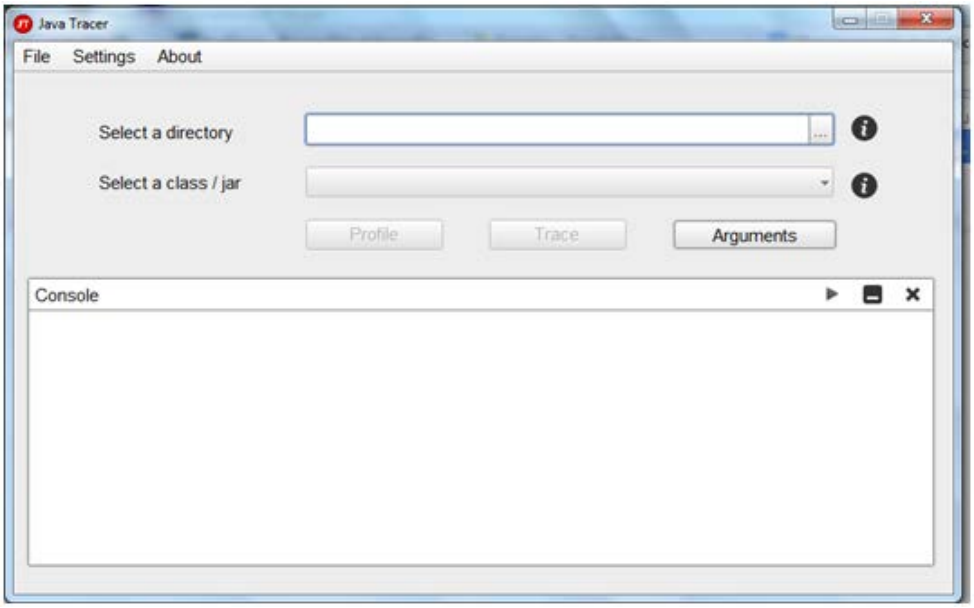


Figura 3.1: Pantalla de inicio

Lo primero que se tiene que hacer es elegir el directorio donde se encuentre el fichero class o jar que se desea analizar. Se pueden dar dos situaciones:

1. Que para analizar el programa se necesite dar información al método principal (*main*), para ello se debe utilizar el botón *Arguments* e introducir los parámetros necesarios. De esta forma se añaden; En el caso de que no sean correctos también se pueden borrar.

2. Si no se necesita dar información extra, se sigue con la ejecución de la aplicación.

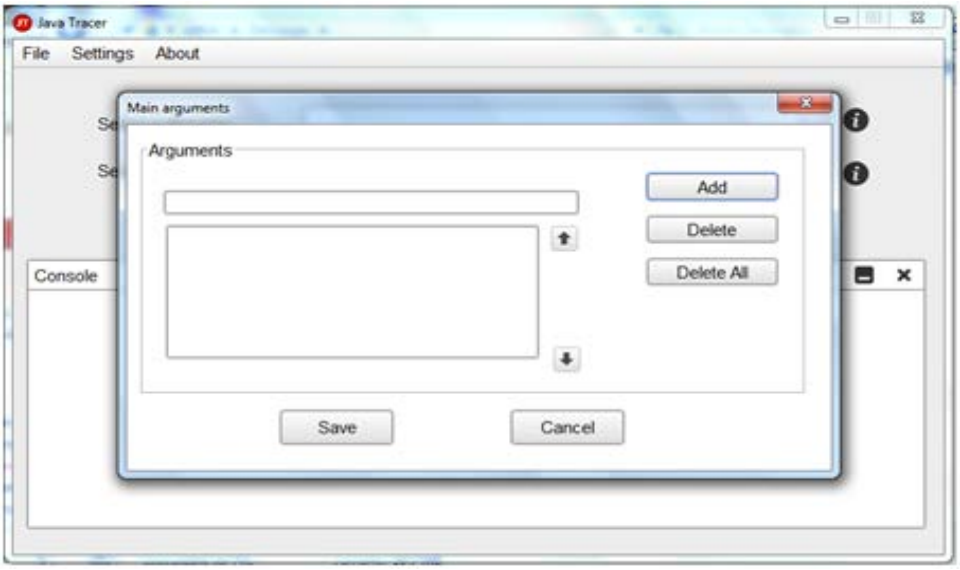


Figura 3.2: Pantalla *Arguments*

A continuación se puede acceder al **Profiler** o ver uno hecho anteriormente, hacer la traza (**Tracer**) y ver la traza que se acaba de hacer o una ya realizada **(Inspector**). Las tres funcionalidades que se acaban de citar son tres módulos independientes entre sí, que necesitan otro módulo para permitir la comunicación entre estos. Estos módulos tienen distintas funcionalidades:

Profiler

Es el análisis de comportamiento de un programa usando información reunida en el propio módulo. Su objetivo es averiguar el número de veces que se han ejecutado las diferentes partes del programa (metodos, clases y paquetes), para optimizar el rendimiento del módulo **Tracer** tanto en consumo de recursos como en la velocidad del proceso, además de poder detectar algunos posibles puntos problemáticos.

Para explicar el funcionamiento del **Profiler** se va a utilizar un programa que implementa el algoritmo *MergeSort*.

Cuando se selecciona la opción **Profiler**, se muestra la siguiente pantalla:

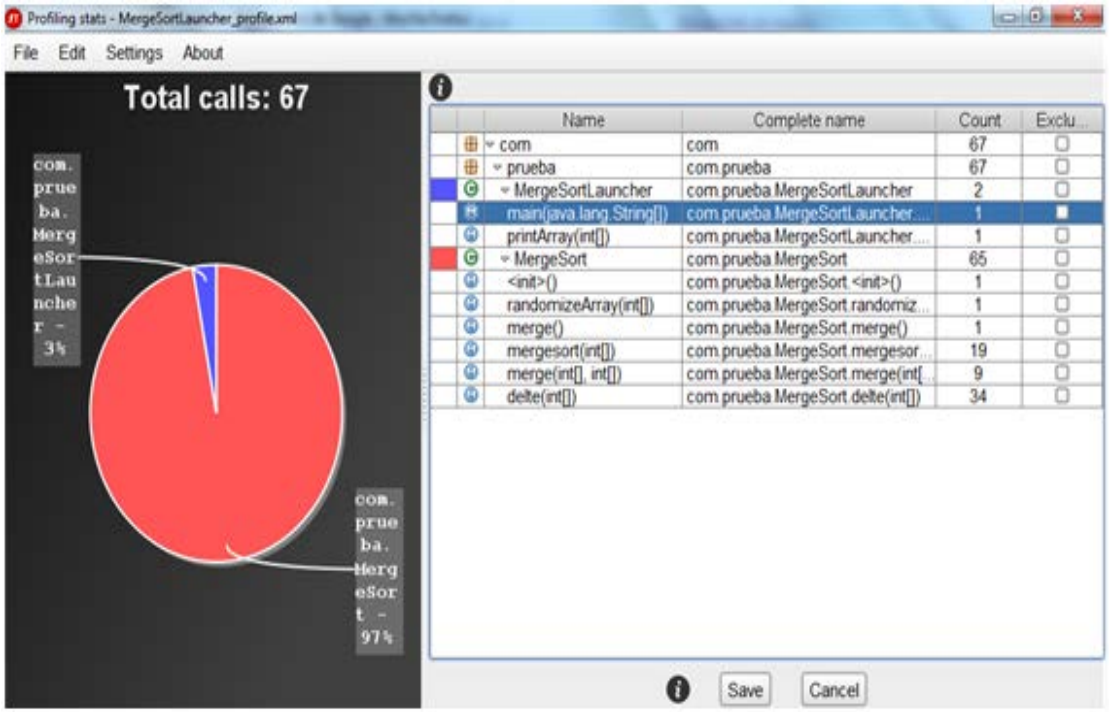


Figura 3.3: Pantalla **Profiler**

En la parte izquierda de la Figura 4.3 se muestra un gráfico con el porcentaje de las clases más utilizadas en la traza para que se puedan excluir las clases menos importantes e interesantes para que el módulo **Tracer** funcione de manera más rápida y eficiente. Como puede verse hay 67 llamadas, de las cuales son el 93% *MergeSort* y un 3% *MergeSortLauncher*.

En la parte derecha se puede ver una tabla donde aparecen (de izquierda a derecha):

- Las clases diferenciadas por colores que aparecen en el gráfico
- Diferentes iconos donde se diferencia si es una clase, método o paquete.
- Los nombres de estas clases, de los métodos que usan y los paquetes a los que pertenecen.
- El nombre completo de los métodos, clases y paquetes del punto anterior.
- El número exacto de veces que se ha llamado al método,clase o paquete.
- ***Excluded*****:** Son *checkbox* que se pueden seleccionar si se quiere ver como quedaría el gráfico al excluir alguna clase, paquete o método. Para guardar estos cambios se usa el botón *Save*.

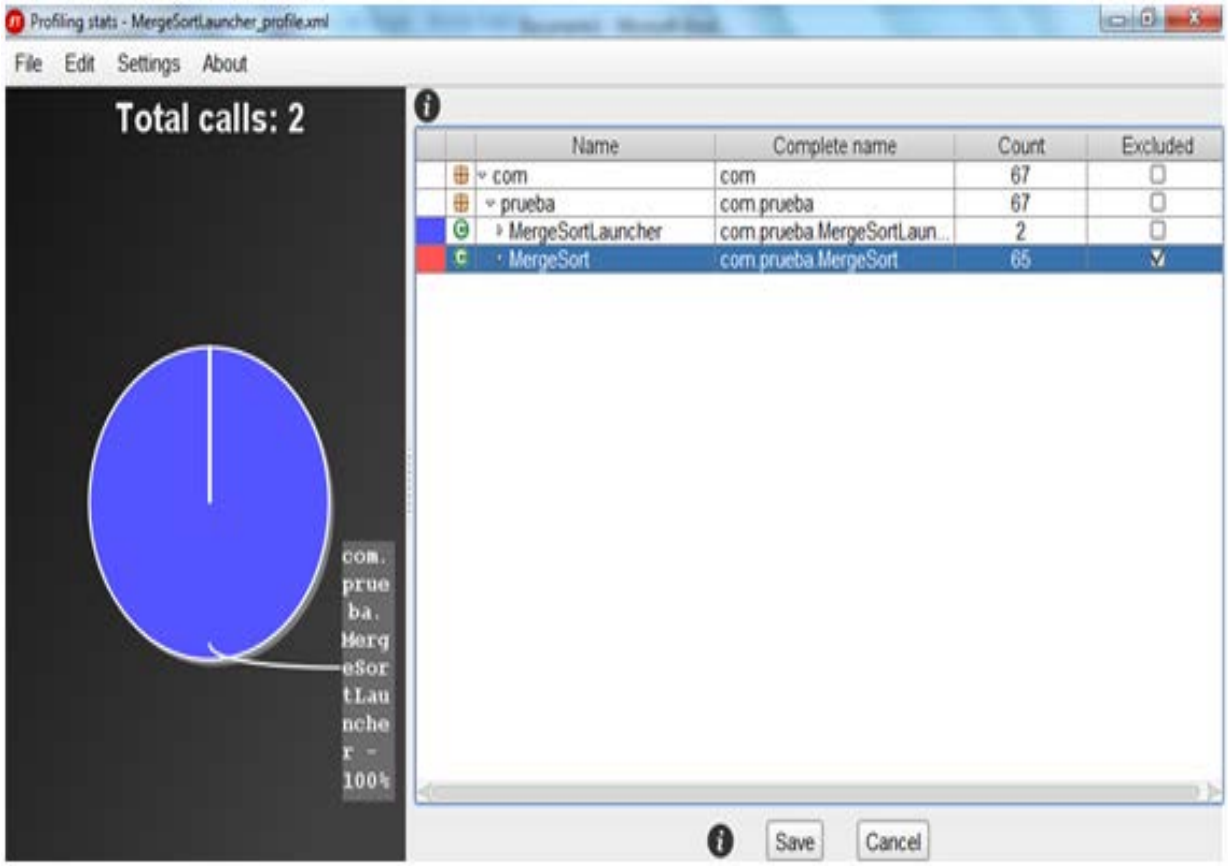


Figura 3.4: Pantalla **Profiler**

En el ejemplo del programa *MergeSort*, como solo se usan dos clases se puede ver en la Figura 4.4 que al excluir una de ellas todas las llamadas pertenecen a la otra. Si se realiza doble click sobre alguna de estas clases en la tabla se abrirá una ventana con otro gráfico, esta vez con los métodos más usados por esta clase en concreto.

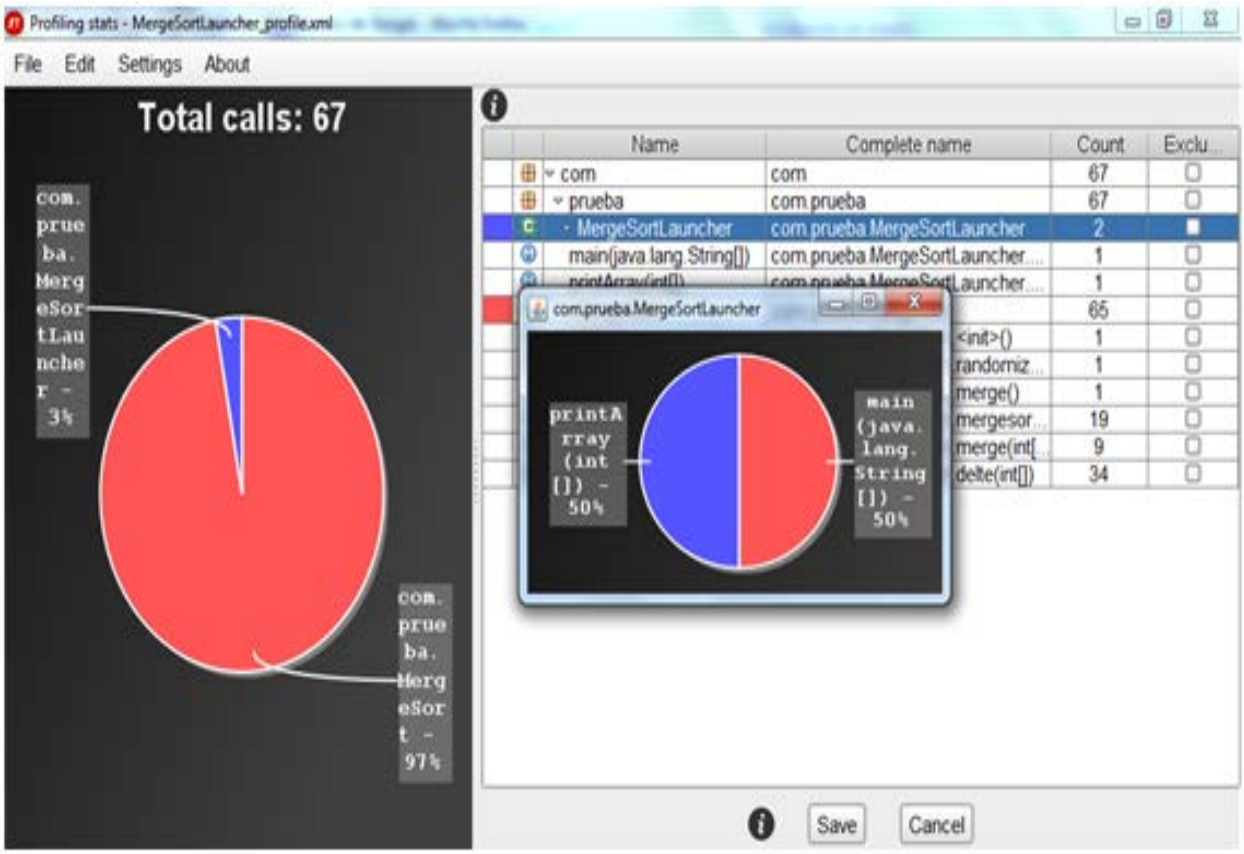


Figura 3.5: Pantalla **Profiler** métodos de una clase en concreto

Cada vez que se ejecuta el **Profiler** se guarda un fichero xml con el que después se podrá abrir un *profile* realizado con anterioridad sin necesidad de volverlo a ejecutar.

Por último también se puede exportar el *profile* como una imagen.

Tracer

Consiste en realizar una traza de un programa usando información que se genera dinámicamente con el valor de las variables del objeto *this* o los parámetros de entrada/salida de los métodos llamados, además de las dependencias entre estos.

Para explicar el funcionamiento de **Tracer** se continuará con el ejemplo anterior. Si se selecciona el botón **Tracer**, se ve la siguiente pantalla:

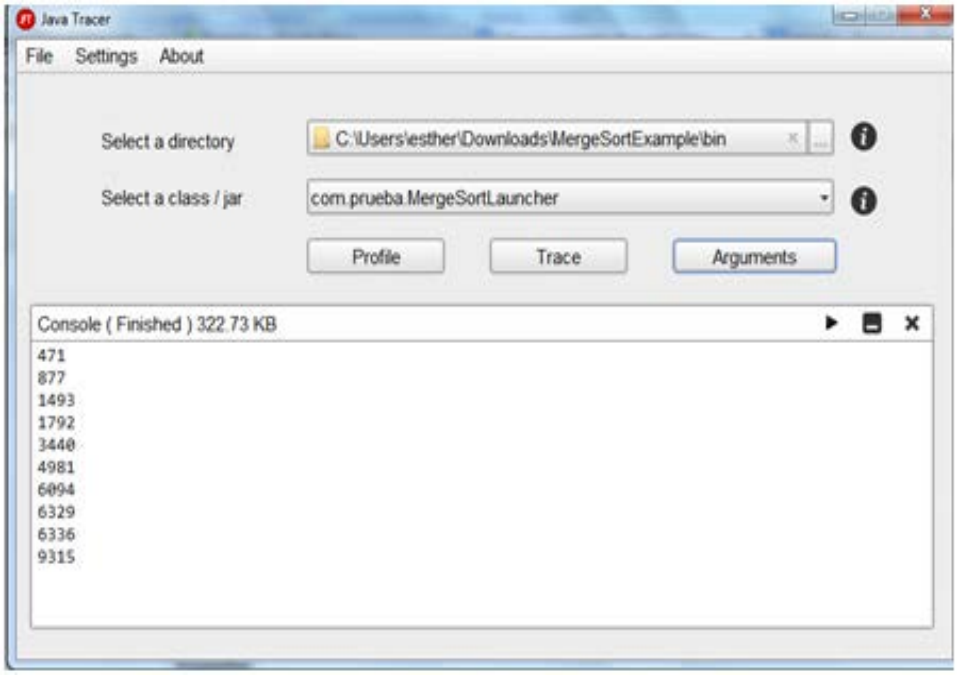


Figura 3.6: Pantalla **Tracer**

En la Figura 4.6 aparecen los valores del array una vez ordenado por el programa que lo genera de forma aleatoria. Lo que hace internamente es generar la traza de la ejecución en un fichero xml, para que se pueda volver a abrir sin necesidad de ejecutar.

Si el programa elegido utiliza entrada/salida o interactúa con el usuario se utilizará la consola de la aplicación igual que en **Profiler**.

Inspector

Este módulo permite visualizar, en forma de árbol, las dependencias entre métodos guardadas por el módulo **Tracer** en su archivo xml.

Siguiendo con el ejemplo anterior, se observa que los nodos del árbol representan las llamadas a métodos, mientras que en cada nodo aparece el nombre del método, así como los parámetros que recibe.

Si se hace doble click en uno de los nodos se puede expandir o contraer el resto de nodos que cuelgan de él para así poder seguir el árbol de llamadas poco a poco y de manera más cómoda para el usuario, en vez de ver un árbol grande con muchos nodos desde el principio (sobre todo si se quiere analizar aplicaciones grandes o con muchas llamadas a métodos).

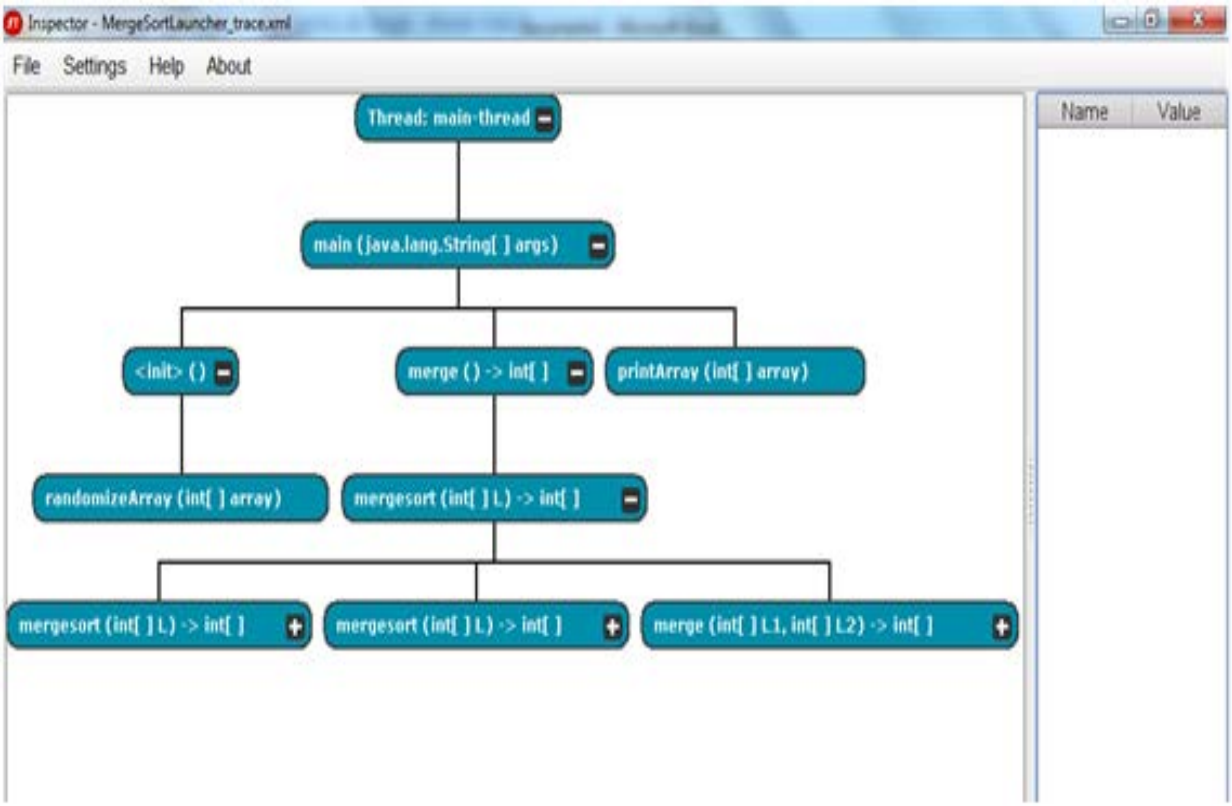


Figura 3.7: Pantalla **Inspector** I

En este caso se muestran diez nodos.

- La raíz (*Thread*) que representa el hilo en el que esta la ejecución.
- El *main*, que es el método con el que empieza la ejecución.
- Tres nodos que son llamados por el nodo *main*. A su vez cada nodo llama a otro y así sucesivamente hasta completar todo el árbol.

Al hacer un click sobre el nodo que se quiere analizar. Se muestra en una tabla el objeto *this* y los valores que devuelve el método o nodo que se ha seleccionado junto con los parámetros de entrada, además de la variable de retorno con su valor (tabla de la derecha Figura 3.8).

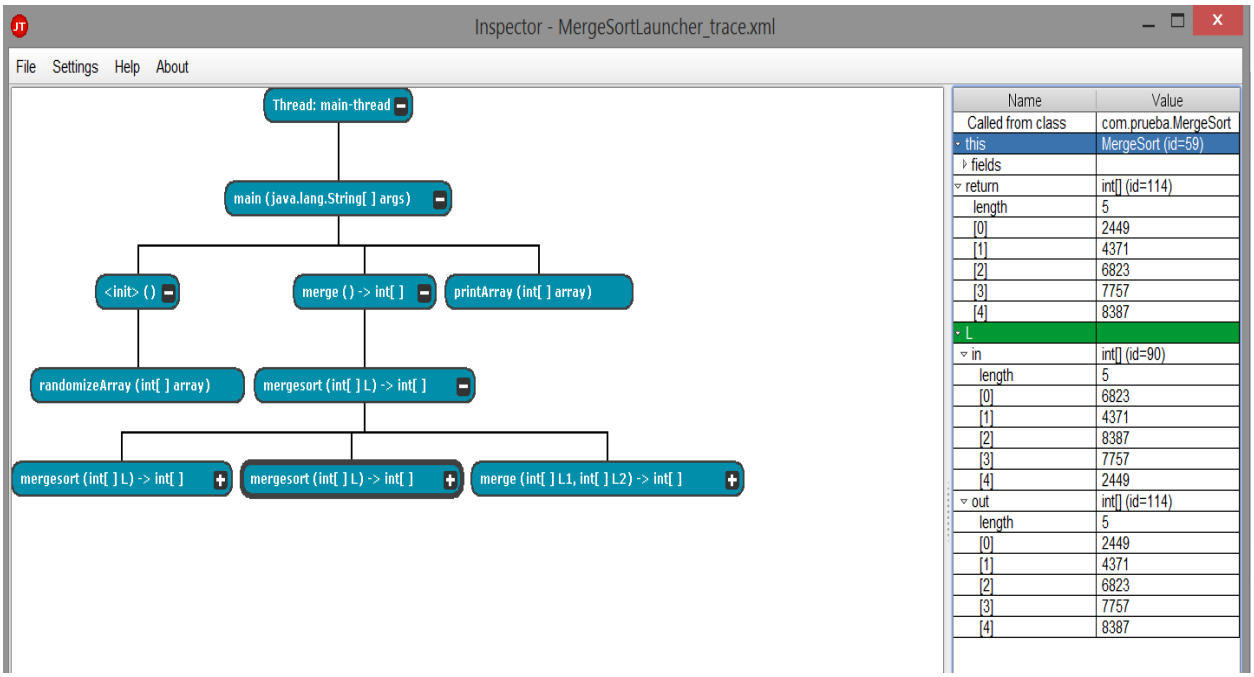


Figura 3.8: Pantalla **Inspector** II

En el caso de que el objeto *this*, los parámetros y/o el valor de retorno hayan sufrido modificaciones en el transcurso del método aparecerán en la tabla de color verde y se mostrarán junto a su antiguo valor para poder comparar los cambios realizados y ver si el método ha hecho lo que se espera o no. Si aparece el nodo del árbol de color rojo, el método devuelve una excepción no capturada y se propaga hasta la raíz.

Con el ejemplo se va a analizar los cambios producidos por una de las llamadas al método *MergeSort*. Se puede ver que a este método se le paso como dato de entrada una array con los siguientes valores: L= [6823, 4371, 8387, 7757, 2449] y cambiaron a L = [2499, 4371, 6823, 7757, 8387]. El método ha hecho lo que se esperaba y ha ordenado el array de menor a mayor.

Menu File

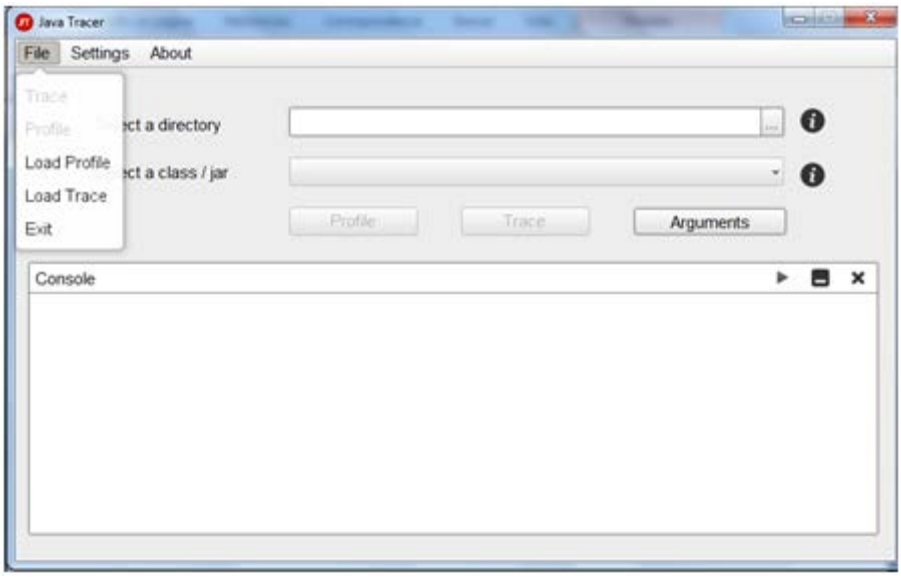


Figura 3.9: Pantalla **Menu File**

Este menú proporciona las siguientes opciones:

- ***Load Profile*:** Carga un archivo *profile* ya existente donde se podrán hacer todas las operaciones del módulo **Profiler**, sin tener que ejecutar este .

- ***Load Trace*:** Carga una traza anteriormente guardada, para poder volver a visualizar el árbol, sin necesidad de volver a ejecutar una traza del programa a analizar.

Settings

Se puede cambiar la configuración por defecto de la herramienta utilizando los *settings*. Estos se pueden dividir en dos, uno que será la configuración del **Tracer** (*Tracer*) y otro para la configuración del **Inspector** (*Display Tree*).

Settings - Tracer

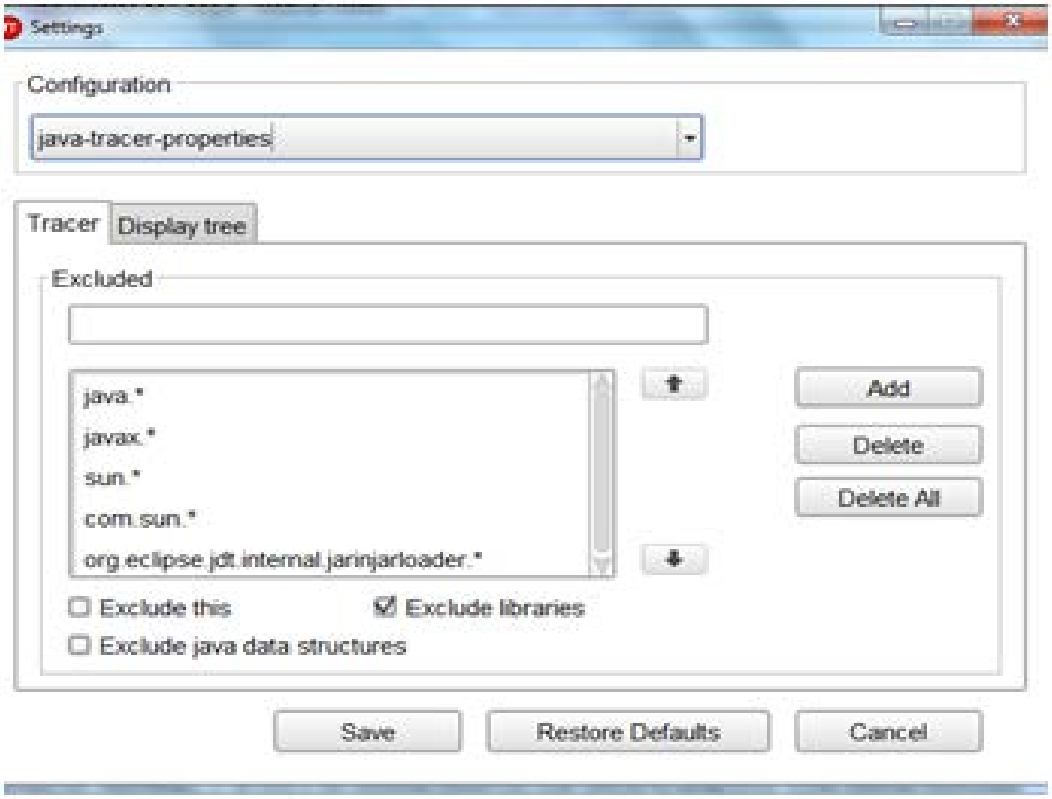


Figura 3.10: Pantalla ***Settings - Tracer***

El campo *Excluded* sirve para introducir (igual que en **Profiler**) las clases, paquetes y/o métodos que se quieran excluir además de los ya excluidos.

Como se observa en la Figura 4.10, por defecto el programa excluye los paquetes que aparecen en la cuadro de texto de la izquierda. El motivo por el que los excluye es por ahorrar recursos y ganar velocidad sin perder información, ya que, estos paquetes son paquetes de Java predefinidos que no suele ser necesario depurar. No obstante si alguno de los paquetes ya excluidos se quiere visualizar en la traza se podrían volver a incluir (también los excluidos por defecto). Para ello, solo haría falta seleccionar el paquete que se quiere incluir y dar al botón "*delete*".

Por otra parte en la parte de abajo de la imagen se observa que hay otras posibilidades de excluir aparte de excluir paquetes, clases o métodos. También se pueden excluir:

1. El objeto *this* de cada clase, en este caso se perdería información pero se ganaría en recursos.

2. Las bibliotecas, que está marcada por defecto por el mismo motivo indicado anteriormente.
3. Las estructuras de datos de Java (ArrayList, HashMap…); Igual que en el primer apartado se perdería información pero se ganaría en recursos.

Settings - Display Tree

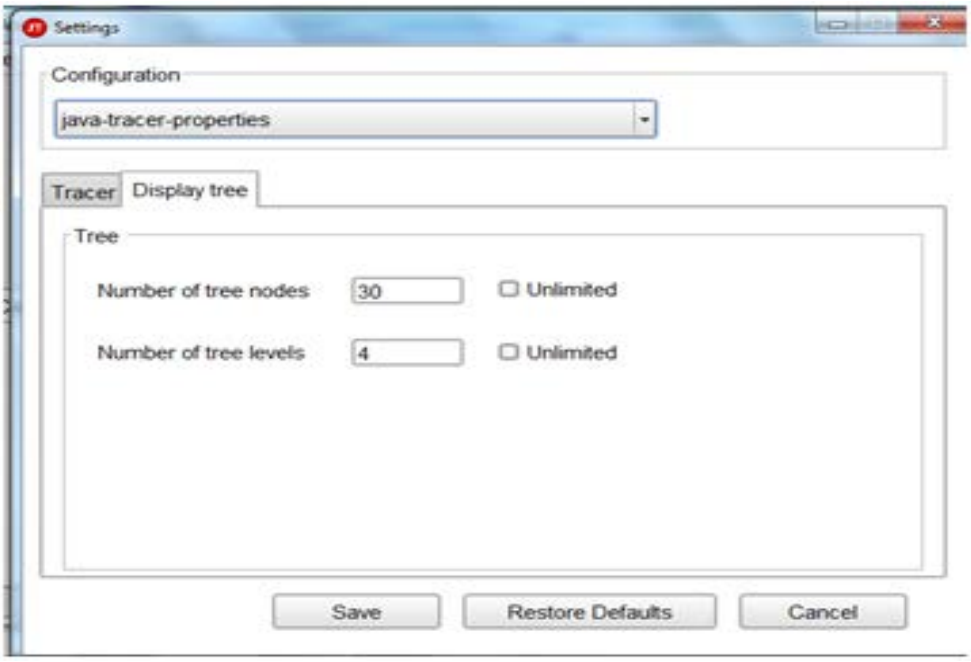


Figura 3.11 : Pantalla **Settings - *Display Tree***

Por defecto se muestran treinta nodos y cuatro niveles, la razón es facilitar al usuario la utilización del árbol, pudiendo desplegar poco a poco los valores que le interesen en lugar de que aparezca el árbol completo. El número de nodos y de niveles se puede alterar seleccionando el número que se quiera o poniendo *unlimited* en cuyo caso aparecerán todos los nodos.

3.2. Limitaciones

Hay que tener en cuenta que, como todas la herramientas y aplicaciones, JavaTracer también tiene ciertas limitaciones.

La principal limitación está relacionada con la escalabilidad: Ejecuciones con gran número de llamadas generan inmensos ficheros de traza que requieren también mucho tiempo para ser generado. Vamos a detallar este problema:

- **Generación de traza:** No se llega a problemas de funcionamiento de la aplicación en este punto. El único límite es el dispositivo físico (espacio libre en el disco), y el principal problema desde el punto de vista del usuario es el excesivo tiempo que puede requerir.

- **Consulta de la traza:** Las bibliotecas estándar XPath que se están utilizando fallan en ficheros excesivamente grandes (por encima de 300 Mb).

Esta limitación puede ser superada al menos parcialmente mediante el uso del **Profiler**. En esta herramienta sólo se apuntan la cantidad de llamadas que se hacen sin generar traza, lo que la hace sumamente escalable (las estructuras internas utilizadas -árbol de clases, paquetes y métodos asociados- tienen una estructura limitada por el diseño del programa objeto, no dependen de la ejecución concreta). Tras utilizar el **Profiler**, el usuario puede ver qué clases son utilizadas más a menudo y excluirlas antes de usar el **Inspector**. A menudo se trata de clases asociadas a bibliotecas externas y por tanto al excluirlas no se pierde información. En otros casos, cuando se trata de clases/paquetes del propio usuario, lo que se puede hacer es excluir inicialmente algunas más “pesadas” y en una posterior ejecución examinar sólo éstas, limitando los métodos a inspeccionar. Para facilitar este uso la aplicación permite salvar y recuperar distintas configuraciones (herramienta *Settings*).

3.3. Casos de estudio

Para demostrar el correcto funcionamiento de la aplicación se quiso utilizar una serie de casos de estudio o *benchmarks* ajenos al proyecto. Dicho casos de estudio se consiguieron mediante comunicación personal con Josep Silva los cuales están en la siguiente página:

http://users.dsic.upv.es/~jsilva/DDJ/experiments.html

No obstante los casos de estudio se pueden conseguir en la página web del proyecto:

https://sites.google.com/site/javatracer/benchmarks

A continuación se enumeran los test usados y una explicación de los mismos:

- **Test 1** - *NumReader*: Realiza la conversión de números enteros a su respectivo en letras.

- **Test 2** - *Orderings*: Se encarga de crear aleatoriamente una lista de diez numeros y analiticamente le realiza una ordenación mediante el método quicksort, mostrando los tiempos transcurridos.

- **Test 3** - *Factoricer*: Añade valores a un vector desde una función que genera números aleatorios, los números generados aleatoriamente son factorizados, el resultado de cada factorización se verifica si es número par, en caso contrario sería impar.

- **Test 4 –** *Sedgewick:* Se ejecuta en tiempo proporcional a N log N, independiente de la distribución de insumos. El algoritmo se guarda en el peor de los casos al azar para revolver los elementos antes de la clasificación. Además, utiliza el método de Sedgewick la que se detiene en la igualdad de llaves. Esto protege contra los casos que hacen que muchas implementaciones de libros de texto, incluso al azar, cuadráticos (por ejemplo, todas las claves son las mismas).

- **Test 5 -** *Clasifier:* Se encarga de crear una cola y llenarla con números aleatorios y, posteriormente, generar otra cola a partir de los números negativos generados en la primera.

- **Test 6** - *LegendGame*: Se encarga de simular el juego de rol LEGEND, mediante la simulación del tirado de dados y el guardado de alguna cantidad igual o menor que la tirada.

- **Test 7** – *Cues:* Se encarga de crear dos colas de números de manera aleatoria y además las ordena, compara y muestra el resultado de dicha comparación.

- **Test 8** - *Romanic:* Añade elementos con la fórmula (i*100)/2+i, la función principal es convertir números a su equivalente en número romano.

- **Test 9** – *FibRecursive*: Clase que realiza la serie de fibonacci a los elementos insertados en un vector. Cabe señalar que los números ingresados son aleatorios y se toma precaución de no repetir ningún número. El último número de la serie es tomado para sacar el factorial correspondiente y verifica si es numero par o impar. Finalmente, se realiza un ordenado de los números.

- **Test 10** – *Risk:* Se encarga de simular mediante tiradas de dados las posibles combinaciones de las tiradas de dados del juego de mesa Risk.

- **Test 11** - *FactTrans*: Llena un vector con valores pares del 10 al 20. Obtiene los factoriales de cada uno de los elementos del vector. Voltea los dígitos obtenidos en el factorial, ejemplo 45689 → 98654. Suma los dígitos contenidos en la cantidad del factorial 4+5+6+8+9 = 32.
- **Test 12** - *RndQuicksort*: Llena el vector con números aleatorios del 1 al 10. Al ir llenando el vector se verifica que los números no sean repetidos. De igual manera va convirtiendo los números aleatorios generados en su representación binaria. Se verifica si el vector está ordenado y de no ser así se ordena utilizando el método Quicksort recursivo.

- **Test 13** - *BinaryArrays*: Se llena el vector con números aleatorios del 1 al 10. Conforme se llena el vector con números aleatorios el programa revisa si el número es par o impar, si es par se obtiene el factorial de este número, de lo contrario se obtiene la cuarta potencia del número aleatorio. De igual manera va convirtiendo los números aleatorios generados en su representación binaria. Finalmente el vector se ordena utilizando el método Quicksort recursivo.

- **Test 14** - *FibFactAna*: Inicializa una cadena con los valores “CAT” y obtiene sus anagramas de forma recursiva. Conforme se obtienen los valores del anagrama se convierten los caracteres a enteros, se almacenan en un arreglo de enteros y se imprimen ambos: el anagrama y sus valores ASCII. A continuación se verifican los valores del arreglo entero de la siguiente forma, si es un número par se obtiene el factorial de forma recursiva, de lo contrario se obtiene la serie fibonacci también de forma recursiva.

- **Test 15** – *NewtonPol*: Calcula raíces de polinomios por el método de bisección y el método de newton.

- **Test 16** - *RegresionTest*: Realiza el cálculo de la media, varianza, desviación estándar y la recta de regresión.

- **Test 17** - *BoubleFibArrays*: Realiza multiplicaciones, fibonacci y ordenamiento burbuja, las multiplicaciones la realiza de dos formas, recursivamente e iterativamente, igualmente fibonacci y el ordenamiento burbuja lo realiza de cuatro formas diferentes.

- **Test 18** - *ComplexNumbers*: Se realizan cálculos con número complejos.

- **Test 19** - *StatsMeanFib*: Realiza el cálculo de la mediana de un arreglo de número, pero para obtener este dato, realiza un ordenamiento mediante el algoritmo quicksort.

- **Test 20** - *Integral*: Integra de dos formas distintas.

- **Test 21** - *TestMath*: Suma los elementos de un vector y calcula el cuadrado del resultado de tres formas distintas. El vector de entrada es: [1, 2]

- **Test 22** - *TestMath2*: Suma los elementos de un vector y calcula el cuadrado del resultado de tres formas distintas. El vector de entrada es: [3, 3, 12, 3, 12, 6, 3, 4, 1, 2].

- **Test 23** - *Figures*: Se crea un grupo de figuras donde se le insertan figuras, se obtiene información de ellas y finalmente se ordena el grupo de figuras.

- **Test 24** - *FactCalc*: Calculadora que suma, resta, divide, multiplica y calcula factoriales usando solo la suma y la resta como operaciones.

- **Test 25** - *SpaceLimits*: Se crea una zona llamada Planta_Baja, un límite, le añade vértices y declara ese límite como el límite exterior de la zona anteriormente creada. Luego calcula si las coordenadas 12,790 (tomadas desde el origen de coordenadas de la zona Planta_Baja) están dentro de la zona (es decir, si están dentro de su límite exterior (el antes creado). Y lo muestra por pantalla (true si están dentro y false si no).

- **Test 26** - *Binary Tree*: Crea un árbol binario a partir de números.

- **Test 27** - *Dijkstra*: Utiliza el algoritmo de Dijkstra para encontrar el camino más corto entre dos puntos.

- **Test 28** - *Figures*: Utiliza figuras para jugar al tres en raya.
- **Test 29** - *Mergesort*: Algoritmo de ordenamiento Mergesort.
- **Test 30** - *Pi-Gauss*: Calcula el número de Pi por el método de Gauss.
- **Test 31** - *Prime Numbers*: Calcula los primeros números primos.
- **Test 32** - *Sort*: Algoritmo de ordenamiento Quicksort.
- **Test 33** - *SQRT*: Calcula la raíz cuadrada utilizando tres tipos de bucles.
- **Test 34** - *Tic-Tac-Toe*: Repite una partida de tres en raya almacenada en un archivo.

4. Conclusiones

Tras el desarrollo y las pruebas, se ha comprobado que la aplicación es útil para localizar errores en muchos casos. Sin embargo, esta herramienta no pretende sustituir al depurador tradicional (como el depurador de Eclipse u otros) sino que se pueda usar como complemento de estos.

Educación

Un posible campo de aplicación de esta herramienta es en el campo de la educación. Ver de forma gráfica el árbol de llamadas ayuda a comprender cómo funciona (o no) el código de un programa Java.

El árbol puede servir para comprender la recursividad de los métodos viendo cómo el mismo método se llama a sí mismo pero con distintos valores hasta llegar al caso base (representado en el árbol como un nodo hoja). De la misma manera, también puede servir para comprender de manera gráfica el *backtracking,* viendo las llamadas que se han producido y el valor de sus argumentos. En el caso de las estructuras iterativas se puede ir observando cómo van cambiando las distintas llamadas en función de la iteración ayudando a explicar los casos extremos (primera y última iteración) y un caso en concreto (casos intermedios). Finalmente, también ayuda a la comprensión de los eventos que se producen durante el flujo de un programa. Esto se debe a que un evento puede aparecer en cualquier parte del flujo. En el caso del árbol, el método sería hijo del método donde ha ocurrido el evento y por ello ayudaría a entender el concepto de interrupción de un programa y suceso de un evento.

En el caso de la Programación Orientada a Objetos, ayuda a comprender las relaciones entre clases. En el árbol, aparte de ver las llamadas, se puede ver qué clase ha sido la que ha llamado a un método. De esta manera, observando el árbol de una manera más global, se puede sacar conclusiones de las relaciones entre clases, dependencias, etc.

También puede ser útil en el ámbito educativo para ayudar a comprender el funcionamiento del control de excepciones con *try-catch-finally.* Cuando se produce una excepción se propaga hasta donde se capture o bien hasta el nodo raíz (normalmente el método main). A lo largo del árbol se observa la propagación de la excepción mediante el coloreado en rojo de los métodos que capturan dicha excepción, motivo por el que se cree que es una buena metodología para entender el funcionamiento de las excepciones en Java.

Finalmente, es importante mencionar el aspecto educativo que puede tener a la hora de explicar el paso de argumentos por referencia o por valor. Esto se debe a que si una variable ha cambiado su referencia se verá cómo al desplegar la variable tiene un campo *in* y otro campo *out* que nos ayuda a diferenciar que ha habido un paso por referencia ya que el id del elemento ha cambiado. Sin embargo, si únicamente ha cambiado el valor se verá el valor antiguo y el valor actual en la columna de *Value*.

Depuración de errores

Otro posible campo de aplicación de esta herramienta es para la depuración de errores, puesto que ayuda a localizarlos de una manera más visual. Una forma de encontrar errores efectiva es comprobar si el método que ha sido seleccionado devuelve el valor esperado, en caso contrario se observa de la misma manera el estado de los hijos y pueden darse dos casos:

- **Uno o más hijos no devuelvan el valor esperado:** Esto quiere decir que el error se encuentra en los descendientes que no han devuelto el valor que se esperaba.

- **Todos sus hijos devuelvan el valor esperado:** Esto quiere decir que el error se encuentra en el método actual.

También se puede observar los cambios de los valores a la salida de los métodos, dichos cambios se visualizan de una manera más rápida, puesto que se pinta de color verde cuando se ha producido un cambio.

4.1. Trabajo futuro

Si se quisiese depurar un proyecto a gran escala con esta aplicación, habría que hacer algunos cambios para hacer frente a las limitaciones de las que se hablaron en el capítulo 3. Limitaciones que se podrían mejorar en futuras versiones de la herramienta. A continuación se mencionan algunas mejoras que llevándose a cabo podrán tener un gran impacto en la aplicación.

Uso de bases de datos

Como se explica en el capítulo anterior, las bibliotecas estándar **XPath** que se están utilizando fallan en ficheros excesivamente grandes (por encima de 300 Mb). Esto se debe a que las bibliotecas, usadas entre otras cosas para extraer un nodo del árbol xml, recorren todo el árbol en busca del nodo que se quiere extraer.

Por lo tanto, cada vez que se construye el árbol de llamadas (en **Inspector**) , o cada vez que se selecciona o se expande un nodo de ***Inspector*** se tiene que recorrer todo el árbol xml para acceder a la información del nodo requerido haciendo este proceso excesivamente costoso, sobre todo en trazas grandes.

Una mejora en la aplicación consistiría en usar una base de datos relacional en vez de ficheros xml. De forma que cada vez que se quisiese la información de un nodo se hiciese una consulta a la base de datos. Así se ahorraría el coste de recorrer todo el árbol xml en busca de un nodo.

Estructuras iterativas

Al realizar una traza puede ocurrir que un método contenga un bucle que en cada iteración llame a su vez a otro método.

Si esto ocurriese el árbol de llamadas de **Inspector** podría ser excesivamente ancho (dependiendo del número de iteraciones) ya que aparecería un nodo que tendría un hijo por cada iteración del bucle.

Para evitar esto, se podría detectar esta situación e introducir un nodo por cada bucle de este tipo. De forma que si un bucle llama en cada iteración al mismo método no se tenga un árbol excesivamente ancho sino que se tenga un nodo contraído de tal forma que si se quisiese ver las llamadas a métodos realizadas en ese bucle se pudiera expandir.

Multithreading

Otra de las ampliaciones que se podrían llevar a cabo en un futuro sería adaptar la aplicación para programas *multithread*, puesto que hoy en día existe un campo muy amplio que requiere la utilización del *multithreading* en sus aplicaciones, con el objetivo de acelerar el tiempo de ejecución y por lo tanto mejorar en el rendimiento.

"La mayoría de los procesadores actuales están diseñados para minimizar el tiempo de ejecución de una única tarea, poniendo el énfasis en técnicas para reducir el tiempo medio de ejecución de las instrucciones y sin considerar demasiado la eficiencia con que se aprovechan los recursos del procesador. Existe la presunción, más o menos implícita, de que la mejor forma de ejecutar varias tareas diferentes es dedicar todos los recursos del procesador a una única tarea, e ir intercambiando las tareas en ejecución a lo largo del tiempo. Sin embargo, cuando se pretende ejecutar varias tareas concurrentemente, la medida fundamental de las prestaciones del sistema ya no es el tiempo de ejecución final de cada tarea sino la cantidad de tareas realizadas por unidad de tiempo o productividad (*throughput*). En este caso, la utilización eficiente de los recursos resulta crucial." [1]

Inspección de métodos

Para terminar de completar JavaTracer se había contemplado la opción de añadir *zoom* a los nodos de **Inspector**. Con esto se conseguiría obtener información mucho más detallada y concreta únicamente del nodo que interesa inspeccionar. Al generar una traza sólo de un nodo en concreto, existiría sólo un fichero con información exclusiva de todo el flujo de dicho nodo.

Con esta forma de inspeccionar el nodo(método) se podría entrar más en detalle, hasta tal punto de no solo poder observar los cambios de los valores de las variables, los valores de retorno del nodo, sino que también se tendría una visión a nivel de instrucciones. Esto sería de gran ayuda al momento de detectar de una manera más fácil y rápida, ya no solo errores de ejecución, sino que también errores dentro de instrucciones que a simple vista no son fáciles de detectar.

Mejora de la navegación en el árbol

La herramienta **Inspector** es probablemente la más utilizada a la hora de usar la aplicación, ya que permite visualizar la traza. Esto se debe a que se invierte el tiempo para la búsqueda de errores a través del árbol de llamadas. Para poder mejorar la eficacia de localización de errores, hay una serie de cambios que podrían mejorar la navegación del árbol de llamadas, facilitando así la labor de depuración:

- **Búsqueda de nodos:** Consiste en la búsqueda de nodos del árbol mediante una consulta. En esta consulta se podría especificar el nombre del método que se busca, qué tipo de parámetros, etc, y que la consulta devolviese el nodo que satisface las especificaciones introducidas. En caso de que exista más de un método que las cumpla, se podría ofrecer al usuario la opción de qué método elegir o afinar la búsqueda. En el caso de que no exista dicho nodo se podría intentar recuperar los nodos más parecidos. Esta opción supondría una gran mejora para la localización de errores, ya que sería mucho más rápido que la navegación en el árbol.

- **Zoom en el árbol:** A veces es demasiado complicado poder observar el árbol en su totalidad. Una posible solución a este problema puede ser mejorar la visualización del árbol utilizando un zoom que permite alejar y acercar el árbol pudiendo así observar más o menos nodos.

- **Mejorar la movilidad:** Puede ser incómodo moverse a través de los *scrolls*. Una solución a este problema puede ser dejar moverse a través de ellos a través de alguna forma de entrada como puede ser hacer *click* en el fondo del árbol y arrastrando el ratón como si se quisiera arrastrar el árbol hacia un lado.

Generación perezosa

Una mejora de rendimiento podría ser la generación del árbol perezosamente. Es decir, se impondría un límite en la profundidad de las llamadas y, si se está interesado en uno de los nodos de la frontera, se podría seguir expandiendo. De esta manera, se ahorraría en espacio y se daría la posibilidad al usuario de continuar la depuración en los nodos que más le interesen.

Desarrollo de la aplicación para el framework .NET

Se podría utilizar .NET para desarrollar la herramienta en otros lenguajes de programación además de Java.

"El *framework* .NET es una plataforma que distribuye el software en forma de servicios que pueden comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma y lenguaje de programación con los que hayan sido desarrollados. Se caracteriza por disponer de un entorno común de ejecución para lenguajes" [2].

"Este *framework* facilita el desarrollo de lenguajes. Su marco de trabajo soporta ya más de veinte lenguajes de programación y es posible desarrollar cualquiera de los tipos de aplicaciones soportados en la plataforma con cualquiera de ellos, lo que elimina las diferencias que existían entre lo que era posible hacer con uno u otro lenguaje" [3].

Algunos de los lenguajes desarrollados para el marco de trabajo .NET son: C#, Visual Basic .NET, Delphi, C++, F#, J#, Perl, Python, Fortran, Prolog , Cobol y PowerBuilder". [4]

2. Bibliografía

[1] *Multithreading en Entornos de Ejecución Multitarea* http://sedici.unlp.edu.ar/bitstream/handle/10915/24180/Documento_completo.pdf?sequence=1
[2, 3 y 4] *Framework* .NET http://msdn.microsoft.com/es-ES/vstudio/aa496123